

Java学习群

72030155

进群可以免费获取视频教程以及每日免费听老师讲课

万水计算机核心技术精解系列

Java 2 网络协议内幕

[美] Al Williams 著

何雄 等译

中国水利水电出版社

内 容 提 要

本书涵盖了 Java 程序设计中各个层次的网络编程, 是使用 Java 进行网络编程的优秀的指导书。本书的主要内容有: Internet 基础、基础套接字、Telnet、FTP 和 TFTP、SMTP、POP3、NNTP、HTTP、HTTPS 等协议的网络程序设计。内容系统而且全面, 概念清晰, 易于理解, 并且每章都给出了大量的实例及分析。可从中国水利水电出版社网站 (www.waterpub.com.cn) 上下载完整的程序清单。

本书适合于有一定 Java 基础的程序员和高级程序员作为编程指南, 也适合对网络协议感兴趣的程序员阅读。

Original English language edition published by The Coriolis Group LLC, 14455 N. Hayden Drive, Suite 220, Scottsdale, Arizona 85260 USA, telephone (480) 483-0192, fax (480) 483-0193. Copyright © 2001 by The Coriolis Group. Simplified Chinese language edition copyright © 2002 by China WaterPower Press. All rights reserved.

北京市版权局著作权合同登记号: 图字 01-2002-0625

图书在版编目 (CIP) 数据

Java 2 网络协议内幕 / (美) 威廉斯 (Williams, A.) 著; 何雄等译. —北京: 中国水利水电出版社, 2002

(万水计算机核心技术精解系列)

书名原文: Java 2 Network Protocols Black Book

ISBN 7-5084-1152-8

I. J… II. ①威… ②何… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2002) 第 048577 号

书 名	Java 2 网络协议内幕
作 者	[美] Al Williams 何雄 等译
出版、发行	中国水利水电出版社 (北京市三里河路 6 号 100044) 网址: www.waterpub.com.cn E-mail: mchannel@public3.bta.net.cn (万水) sale@waterpub.com.cn 电话: (010) 68359286 (万水)、63202266 (总机)、68331835 (发行部)
经 售	全国各地新华书店
排 版	北京万水电子信息有限公司
印 刷	北京市天竺颖华印刷厂
规 格	787×1092 毫米 16 开本 28.75 印张 622 千字
版 次	2002 年 7 月第一版 2002 年 7 月北京第一次印刷
印 数	0001—5000 册
定 价	48.00 元

凡购买我社图书, 如有缺页、倒页、脱页的, 本社发行部负责调换

版权所有·侵权必究

译 者 序

这是一本实践性和实用性很强的 Java 编程用书。书中的内容主要围绕如何使用 Java 语言运用基本的网络协议进行网络编程。从基本的网络概念到复杂的网络协议剖析，内容涵盖了最基本、最常用的网络协议，还包括对 Java Servlet、Applet、XML 以及 Java 安全性的相关介绍。

本书对网络协议进行了全面的介绍。全书包括 15 章和两个附录。每章基本围绕某类协议进行重点介绍，按照深入介绍、快速解决方案分节进行相关讲解，每章都有详细的程序清单以及对程序的讲解和技巧提示。第 1 章是对 Internet 相关知识的回顾，对基本网络协议的简单总结。第 2 章到第 12 章是对一些基本协议分类进行重点讲解。第 13 章讲解 Java Servlet 的应用。第 14 章是 Java 在 XML 环境下的编程。第 15 章讲解了 Java 的安全性知识。两个附录分别对有用的 RFC 文档以及最常用的端口号进行列表说明，便于读者参考查询。本书中完整的程序清单可从中国水利水电出版社网站（www.waterpub.com.cn）上下载。

本书各章的内容相对独立，如果对网络协议已有相当程度的了解，那么对某些章节可以不必详细阅读，而直接跳到要看的章节查看感兴趣的内容。读者完全可以把本书当作一本 Java 网络编程手册。

Java 作为跨平台的编程语言，非常适合于进行网络编程。如果读者对网络协议只有初步的了解，那么可以详细地参看第 1 章，本章对基本的网络协议都进行了简明的介绍。

本书是为 Java 程序员编写的，也适合于了解一些网络协议的网络编程人员，还适合于希望全面了解 Java 编程的相关人员。

参加本书翻译的除何雄外，还有赵红超、程振林、付岩、王伏生、丰强泽、李东来、张奎、瞿云、刘辉、那艳梅、杨万钦、廖礼萍、魏永新、陈艳华。由于译者水平有限，难免有疏漏之处，敬请读者批评指正。

译 者

2002 年 5 月

致 谢

非洲有一句古老格言是这样说的，“用一个村子来抚养一个孩子”。我不太确信自己是否同意这个观点，但我确实知道出一本书确实需要很多人才行。就像冰山之一角，你只看到了作者。但是，还有编辑、发行人员、排版人员、索引人员和技术编辑，都为最终的发行做出了贡献。没有他们这本书不会出现你的身边，非常感谢他们做好这项工作。还有很多人无法一一列出，但还是要感谢 Kelly、Hilary、Andrew、Cheryl、Peggy、Chris、Kevin 和 Jawahara。

除了 Coriolis 职员外，Mana Tominaga（《Web 技术》杂志的一个同事）为这本书的内容作了大量的工作。谈到《Web 技术》杂志，这本书里的一些程序最开始就出现在我的 Java@Work 专栏里（起码在某种形式上是），要感谢我们的总编 Amit Asaravala 的支持。

最后，如果没有家庭的理解和帮助我不可能写出任何东西。这本书尤其如此，因为它出版期间经历了一段难熬的时光。然而，我可以诚实地说，猎犬 Madison 和波美拉尼亚小狗 Sassy 没有帮多少忙。我儿子的猫 Lacey 更是一点忙都没有帮。

——Al Williams

作者简介

Al Williams 从 Univac1106 机开始就是一位资深程序员。从那时起，他曾经作过 Unix 的编码人员、MS-DOS 程序员和 Windows 开发人员，现在他发现自己最常用的是 Java。Al 还是《Web 技术》杂志中的 Java 和开发专栏作家，他还为其他几个杂志撰写专栏文章，包括《Dr. Dobb's Sourcebook》和《可视化开发》。当 Al 不写文章或者咨询的时候，他就在美国各地讲授计算机编程。在比较少的空闲时间里，他还有无线电这个业余爱好，过去 24 年里他一直有这个业余爱好。他们全家住在休斯敦，德克萨斯附近。

前言

我第一次面对大规模网络像 Internet 可能是 DIALOG。它是一种服务（在 Roger 会议上成立，最终被 Lockheed 购买），允许你采用付费的方式连接不同的数据库。返回到 80 年代初期，访问费用依赖于数据库，但是一个小时要用 100 美元。后来，我工作的公司需要接收电报信息。在看到为接收电报信息来安装它很昂贵时，我们发现可以通过 CompuServ（或者它们的对手 The Source）为 EasyLink 签约，大概 30 美元一个月（没有把每分钟的连接付费计算在内，不过还是要比电报便宜一些）。

当然，我还得用 CompuServ 做实验。我不仅要为每分钟付费，而且还要为在此基础上的远程呼叫付费。在那时没有无限的波动。当然，也没有那么多的内容。记住，这不是 Internet——只是 CompuServ 网络。通过 CompuServ 来访问 Internet 还是很多年以后的事情。

CompuServ 提供了“论坛”，你可以发送各种主题的消息。他们也迷上了另外一些服务，像 EasyLink 和 Saabre（为了预订航线）。我想他们也许提供了实时聊天，但是每分钟要 2 到 3 美元，我付不起这个帐。当然 CompuServ 也能让你下载文件。使用 1200 波特（baud）的调制解调器，你可能也下载不了多少东西。而拨号进入本地的电子公告板系统下载你要的东西就要便宜一些。

今天可就不同了。电子公告板计算机正在减少（或者被转换成 Web 服务）。现在的网民比以前多，他们很多人使用宽带访问，速度超过 1M/s。你可以发现 Web 上有很多令人迷惑的产品、信息和服务。从微不足道开始，网络已经改变了我们生活的每一个角落。

对成熟技术的正确性测试只有等到它完全消失。如果你不太重视技术，这个开始也许不太明显，因为你可能有一个关于你周围有多少事物工作的好的想法。做一个试验，请一些非技术人员解释电是如何进入他们的屋子的，或者电话呼叫连接的距离有多远，这时你将得不到非常好的答案。同时计算机也在很快变成透明的了，20 年以前，一般计算机用户可以告诉你每条指令需要多少个周期、它们计算机上每个芯片的详细情况。今天，绝大多数用户都知道 CPU 是什么类型、制造商声称的时钟速度是多少。

网络也在变向透明，很大部分是由于 Internet 的快速增长。如果你只是近来才接触 Internet，将难以想象它的增长速度有多快。试想想：可以认为 Internet 开始于 1969 年 10 月当 UCLA 的一台机器和 SRI 的一台机器以 50Kbps 的数据速度连在一起的时候。确实，网络看起来并不太像我们今天知道的 Internet——TCP/IP 出现是 8、9 年以后的事了。同时，Internet 还可以追溯到旧的 ARPANet 时代。到 1982 年的时候，网上已经有 200 台主机了。1984 年那个数目已经涨到 1000 了。也就是 15 年里有了 1000 台计算机连在网络上。三年以后的 1987 年，网上主机数达到 10000。到 1992 年，只是 5 年多的时间，Internet 上已经有了超过一百

万台计算机了。最后一次我听说，Internet 上已经有 1600 万台计算机，到 2000 年，游人注册到了第 1000 万个域名。

不久以前，网络是一个神秘的黑盒子，只有一些程序员才需要去考虑。现在，很少有程序不需要与网络交互工作。即使你的程序没有网络功能，你也可以允许对它进行网络安装或者自动升级。

幸运的是，Java 语言具有强大的网络功能——如果你知道如何使用这些功能的话。Java 的一些网络操作是如此简单，那些操作已经变成不透明的了——很像打一个长途电话不可见一样。但是，另外一些网络功能需要小心使用，以避免兼容问题或者引起安全问题。

尽管我们绝大多数人对我们的电、水和电话服务如何工作只有一个模糊的认识，仍然有这方面的专家指导它们的工作细节。如果没有他们，这些基本的服务将不会起作用（接着你会注意到它们）。Internet 也一样。在幕后，一些组织较松散的标准化组织和个人（像已经过世的 Jon Postel），一定程度上控制着 Internet 的技术。就像当前 Internet 将走向何处的政治和历史同样很有兴趣。但是对创业者而言，他们更感兴趣的是这个庞大的网络如何工作好，以每天与世界各地的人相联系。

本书内容

这本书涵盖了使用 Java 的各个层次的网络程序设计。不仅包括高级的技术，像协议的句柄，还包括如何运用底层的套接字对象编写程序，处理下列各种协议：

- Telnet
- FTP 和 TFTP
- SMTP
- POP3
- NNTP
- HTTP
- HTTPS

除此以外，你还能找到关于一些更简单的协议以及多点传送的信息。但是，网络通信只是其中的一部分。数据的格式以及对数据的解释同样重要。你会发现一整章用来介绍如何解释 HTML 和 XML 数据。还会发现关于格式化的邮件消息和其他数据类型贯穿本书始末。

这本书适合我吗

如果你曾经用 Java 编写过程序或者有一定的 Java 知识，这本书将非常适合你，至少让你学会如何进行网络编程。然而，打开套接字只是网络编程的一部分。你不仅要建立与另一台计算机的通信，还要知道在建立连接之后传输什么内容以及如何传输这些内容。

本书将向你介绍在大量常用协议上如何使用 Java 建立客户端和服务端应用。当然，你将学会如何使用套接字。但你一旦建立了与 FTP 服务器、Web 服务器、或者 Email 服务器的

连接，你将学会传送些什么数据以及如何传送数据。

另外有一个 Internet 现象是程序员不用再在“真空”的环境下工作。Internet 上有各种各样的例程和免费的函数库——其中许多就是用 Java 编写的。这可以为你节约大量的时间，因为你可以参考别人的代码来编写自己的程序。

本书比较适合快速学习以尽快取得效果。每章包括一个协议或者协议家族。在“深入介绍”部分都详细地介绍了协议、操作以及实现细节。“快速解决方案”部分提供了你在开发自己代码过程中可能出现的具体问题的解答。并且，每一节还指出对你的工作有帮助的 Internet 上能免费得到的软件。

如果你想综合运用你的 Java 知识来编写常规的网络软件，或者想更多地知道网络协议是如何工作的，那么这本书很适合你。运用这本书里提供的知识信息，你会用 Java 轻松地编写出传输文件、发送 Email、处理网页的程序。

读本书之前应该知道些什么

你应该对简单的 Java 程序设计比较熟悉。如果没有使用套接字编程（或者你没有使用所有类型的套接字编程），会发现第 2 章很有用。并且，该章还涵盖了你可能不太熟悉的其他的 Java 技术（网络编程并不常用的）。

你需要对 Java 知道多少呢？你会很惊奇地发现，你实际上编写各种网络程序只需要很少的 Java 知识。很明显，你需要理解类的概念原理以及静态成员与非静态成员的区别。你应该善于处理基本的语言结构（例如 if、for，以及类似的声明）。

Java 将套接字视为输入输出的另一种方式（来源），因此了解掌握 java.io 库是很重要的。如果你没有使用这个库，就有必要了解第 2 章中的内容。

文本与图形

在本书里，我尽可能避免使用图形用户接口（GUI）。有少数地方，如显示 HTML 用窗口化的接口更易理解，在那些地方你会发现使用 Swing 库的代码。

然而，大量的网络程序实际上不需要图形用户接口。控制台程序更容易理解，并且也是必需的。同时，重点是开发出使用各种协议来工作的对象。如果你有了可重用的对象，你就可以将它用于任何程序——图形的或者其他程序。

使用可重用类的一个例子就是 Java 服务器页面（JSPs）。如果你想看网上的新闻，或者操纵一个探测网关，组合了网络可行的 JSP 程序可以完成这项任务。

为何使用 Java

多年以来，用于开发网络程序的传统语言是 C 和 C++。有很多 C 语言网络程序的例子，大部分的网络教材假设使用 C 语言。

同时，Unix 套接字库是很多其他套接字库的模型，包括微软的 WinSock 库。周围有这么

多 C 语言的例子代码，为何转而用 Java？

有两方面的原因。首先，Java 的库可以随心所欲地扩充，以帮助你编写网络程序。它能成功也因为它是面向对象语言，因为设计者在开始时就有这个目标。

Java 是一种有趣的网络语言的另一个原因是，至少在理论上在任何类型的计算机上写好的 Java 代码能在其他类型计算机上运行。看看 applets（在 Web 浏览器里运行的小程序）就知道。一个 Web 冲浪者可能使用任意一种计算机（Mac，Unix，Windows 甚至其他更古怪的操作系统）。其他系统（像 ActiveX）需要为每一类型的机器写一个单独的程序。而 Java applet 不加改变就可以在支持 Java 的任意机器上执行。

我需要些什么

为了运行书中的所有程序示例，你将需要知道编译 Java 程序的一些方法。我是使用 Java SDK 1.4 版来建立编译书中的程序的，尽管绝大多数也可以在 1.3 版里运行。像 Java 一样，程序不偏向于任何一种操作系统。

当然，你可以选择自己喜欢的不是面向命令行的编译器。如果是这样，在你的操作系统中有多种编译器可供选择：

- CodeWarrior—www.metrowerks.com/desktop/java/
- Forte—www.sun.com/forte/ffj/overview.html
- FreeBuilder—nisoft.orbitel.bg/freebuilder/
- Jbuilder—www.borland.com/jbuilder/
- JCreator—www.jcreator.com/
- Jipe—e-i-s.co.uk/jipe/
- RealJ—www.freejava.co.uk/

如果你在网上作每件事，那你也可以在 www.chami.com/webide/ 上编译你的代码，但我没有试验过。

当然，要从本书的程序中获益最多，你就需要网络。网络上有两三台计算机就可以了，或者有 Internet 连接就更好了。在某些情况下，你可能需要拥有自己的服务器，这样你就不用使用别人的服务器来做试验了。

除此以外，有几章使用了 Sun 公司的附加的库，或者一些开放源码库和工具。这些章将告诉你如何获得你需要的工具。有些章还含有 JSP 代码。要运行这些代码你就需要访问 JSP 容器。幸运的是，有很多免费的 JSP 容器，包括 Tomcat (<http://jakarta.apache.org/>) 和 JRun 的开发人员版本 (www.allaire.com/products/jrun/index.cfm)。在大多数情况下，代码的 JSP 端口是可选的——你还可以不用 JSP 程序来测试这些类。

因为很多程序使用命令行，你可能发现 Windows 命令行提示的局限（特别是 Windows 98 下的命令行没有滚动缓冲）。如果你熟悉 Unix 或者 Linux，你可能宁愿从 <http://sources.redhat.com/cygwin> 下载免费的 Cygwin 包。这个包为 Windows 提供了一个完全

类似于 Unix 的环境。从用户的角度来看，你要有一个 bash 壳以及你期待从 Unix 上得到的所有工具。从开发人员的观点来看，那些并不是很有趣，只有用户环境是值得免费下载的。

开始！

下一步就依赖于你的确切需要了。书后边的一些章节多少有点依赖于前边章节里的内容。所以，最好是顺着章节来看。但是，如果比较急，也可以选择那些你要测试或者用到的协议，直接看它所在的章节。

如果你计划跳着看，你至少可以略掉第 1 章，也可以粗看一下第 2 章，确信自己了解该章讨论的所有 Java 技术。如果发现有不熟悉的地方，应该看完这些章节再看后边的章节。

像所有黑皮书一样，本书的一个特征是在各章里都有快速解决方案，向你提供各个主题的快速总结。如果你特别紧急，可以考虑从快速解决方案开始，参考该章的主要部分。

每一章包含了解释主题的代码清单。为了方便，完整的清单在中国水利水电出版社网站上。

欢迎你对这本书提出意见和反馈。可以发送 email 到 Coriolis 小组的信箱 ctp@coriolis.com。不管你选择什么方法，只看前言学不到很多东西！选择一个主题开始吧。

目 录

译者序	
致谢	
作者简介	
前言	
第 1 章 Internet 基础	1
1.1 深入介绍	1
1.1.1 Java 怎么样	1
1.1.2 协议包	2
1.1.3 Internet 地址	2
1.1.4 DNS: Internet 上的电话簿	3
1.1.5 URL, URI, URN	4
1.1.6 层 (Layers)	5
1.1.7 基本协议	6
1.1.8 网络硬件	7
1.1.9 协议的学习	8
1.2 快速解决方案	9
1.2.1 确定你的真 IP 地址	9
1.2.2 使用动态重定向	9
1.2.3 确定 IP 地址的类型	10
1.2.4 选择端口号	10
1.2.5 使用 Ping 和其他工具	11
1.2.6 自己动手练习一个协议	13
1.2.7 查找 RFC	13
1.2.8 为代理服务器设置 Java	15
第 2 章 Java 网络编程	16
2.1 深入介绍	16
2.1.1 套接字编程的实质	16
2.1.2 I/O (输入/输出) 流	21
2.1.3 高级套接字方法	26
2.1.4 线程	29
2.2 快速解决方案	32
2.2.1 解析主机名	32

2.2.2	向服务器端打开 TCP 套接字	32
2.2.3	打开服务器端套接字	33
2.2.4	创建 UDP 套接字	33
2.2.5	向 TCP 套接字发送数据	34
2.2.6	从 TCP 套接字接收数据	34
2.2.7	压缩套接字数据	35
2.2.8	设定套接字的最长读时间	36
2.2.9	设定服务器端最长接收时间	37
2.2.10	设定 SoLinger	37
2.2.11	设定套接字的延时行为	37
2.2.12	设定保持活动选项	37
2.2.13	设定缓冲区的大小	37
2.2.14	处理套接字异常	38
2.2.15	创建多线程服务器程序	38
2.2.16	自动处理多线程服务器	39
2.2.17	使用线程池为客户端程序服务	41
第 3 章	简单协议	45
3.1	深入介绍	45
3.1.1	Echo 协议	45
3.1.2	Finger	49
3.1.3	Whois 协议	57
3.1.4	基本时间协议 (Basic Time)	59
3.2	快速解决方案	65
3.2.1	使用 Echo 协议	65
3.2.2	编写 TCP Echo 服务程序	65
3.2.3	编写 UDP Echo 服务程序	65
3.2.4	合并 TCP 和 UDP 服务程序	66
3.2.5	使用 Finger 服务	66
3.2.6	编写 Finger 服务器	67
3.2.7	创建一个简单的代理	68
3.2.8	使用 Whois	69
3.2.9	查询对人可读格式的时间	69
3.2.10	对 NIST 时间串进行解码	69
3.2.11	查询机器可读格式中的时间	70
3.2.12	编写时间服务程序	71

3.2.13 选用 Unicode 作字节映射	72
第 4 章 TFTP 协议	78
4.1 深入介绍	78
4.1.1 关于 TFTP 协议	79
4.1.2 Play by Play	81
4.1.3 TFTP 客户端应用	81
4.1.4 TFTP 服务器端应用	85
4.1.5 更简单的一种方法	88
4.1.6 TFTP 与 FTP 的对比	89
4.2 快速解决方案	90
4.2.1 探寻 TFTP 的规范	90
4.2.2 创建一个 TFTP 类	90
4.2.3 创建一个 TFTP 的客户端应用程序	100
4.2.4 创建一个 TFTP 的服务器端应用程序	100
4.2.5 使用 GNU 的 TFTP 类	100
4.2.6 配置 GNU 的 TFTP 服务器	101
第 5 章 Telnet 协议	102
5.1 深入介绍	102
5.1.1 Telnet 回顾	103
5.1.2 NVT 回顾	103
5.1.3 特殊命令	104
5.1.4 要协商的地方	105
5.1.5 Telnet 实践	108
5.1.6 一个基本的 Java 客户端	109
5.1.7 创建一个 Telnet 服务器端应用	114
5.1.8 定制服务器端	120
5.1.9 Telnet 开放源码	121
5.2 快速解决方案	123
5.2.1 探寻 Telnet 协议规范	123
5.2.2 发送 Telnet 命令同时发送数据	124
5.2.3 模拟 NVT	124
5.2.4 协商 Telnet 的选项	124
5.2.5 防止循环	124
5.2.6 处理子选项	125
5.2.7 从基类创建一个 Telnet 客户端	125

5.2.8	从基类创建一个 Telnet 服务器端.....	126
5.2.9	使用 TelnetWrapper.....	127
第 6 章	FTP 协议	128
6.1	深入介绍	128
6.1.1	基础	128
6.1.2	传输	129
6.1.3	响应	130
6.1.4	登录	132
6.1.5	创建连接	132
6.1.6	FTP 命令细节	134
6.1.7	考虑客户端	137
6.1.8	考虑服务器端	138
6.2	快速解决方案	146
6.2.1	查找 FTP 规范	146
6.2.2	连接到 FTP 服务器	146
6.2.3	解释 FTP 的响应	147
6.2.4	管理当前目录	147
6.2.5	读文件目录	148
6.2.6	传输文件	151
6.2.7	选择主动方式还是被动方式	152
6.2.8	使用 FTP 的开放源码	153
第 7 章	SMTP 协议	155
7.1	深入介绍	155
7.1.1	验证	157
7.1.2	超时、多行和透明性	157
7.1.3	扩展的 SMTP	158
7.1.4	题头	158
7.1.5	编码	159
7.1.6	实现	160
7.1.7	使用 SMTP	174
7.1.8	附件	176
7.1.9	SMTP 的问题 (Twists)	177
7.2	快速解决方案	177
7.2.1	探寻 SMTP 规范	177
7.2.2	连接一个 SMTP 服务器	177

7.2.3	通过 SMTP 发送邮件	178
7.2.4	解释响应码	178
7.2.5	形成地址	179
7.2.6	选择题头	180
7.2.7	格式化消息文本	180
7.2.8	使用可引用可打印编码对消息文本编码	181
7.2.9	使用 Base 64 编码对消息文本编码	182
7.2.10	格式化多部分消息	184
7.2.11	使用 MailMessage 对象	185
7.2.12	使用 SMTP 对象	186
第 8 章	POP3 协议	187
8.1	深入介绍	187
8.1.1	POP3 协议	188
8.1.2	一个 POP3 类	190
8.1.3	一个常用列表管理器	190
8.1.4	代码	191
8.1.5	作用	197
8.1.6	进一步开发	197
8.1.7	关于 IMAP	198
8.1.8	使用 JavaMail	198
8.2	快速解决方案	199
8.2.1	探寻 POP3 协议规范	199
8.2.2	探寻 IMAP 规范	199
8.2.3	解释 POP3 服务器的响应	199
8.2.4	使用 POP3 授权	199
8.2.5	了解邮箱状态	200
8.2.6	确定消息细节	200
8.2.7	读一个邮件消息	201
8.2.8	删除一个消息	201
8.2.9	创建一个 POP3 客户端类	201
8.2.10	使用 com.jthomas.pop 包	205
8.2.11	安装 JavaMail	206
8.2.12	使用 JavaMail Message 对象工作	206
8.2.13	使用 JavaMail Session 对象工作	207
8.2.14	在 POP 邮件服务器中使用 JavaMail	207

8.2.15	在 IMAP 邮件服务器中使用 JavaMail	209
第 9 章	NNTP 协议	211
9.1	深入介绍	211
9.1.1	关于 News	211
9.1.2	NNTP 内幕	212
9.1.3	封装 NNTP	215
9.1.4	Web 上的 NNTP	221
9.2	快速解决方案	222
9.2.1	探寻 NNTP 规范	222
9.2.2	连接一个 News 服务器	222
9.2.3	选择一个组	223
9.2.4	列出所有的组	223
9.2.5	寻找新组	223
9.2.6	读取文章	224
9.2.7	改变当前的文章	224
9.2.8	查找新文章	225
9.2.9	投递文章	225
9.2.10	使用 NewsClient 类	225
9.2.11	显示 Web 上的文章	226
9.2.12	读 Web 上的文章	228
9.2.13	通过 Web 投递文章	230
第 10 章	HTTP 客户端	232
10.1	深入介绍	232
10.1.1	HTTP 协议	232
10.1.2	状态码	235
10.1.3	常用标题	236
10.1.4	表单	237
10.1.5	Cookies	239
10.1.6	Applets (小程序)	248
10.2	快速解决方案	257
10.2.1	探寻 HTTP 协议规范	257
10.2.2	创建简单请求	257
10.2.3	创建 1.0 版的请求	257
10.2.4	创建 1.1 版的请求	258
10.2.5	读状态码	259

10.2.6	通过 HTML 向服务器发送表单数据	259
10.2.7	使用 Java 发送表单数据到服务器	260
10.2.8	URL 数据编码	260
10.2.9	自动提交表单	261
10.2.10	发送和接收 Cookies	262
10.2.11	打开浏览器到浏览器的通信	263
10.2.12	检查合法链接	263
第 11 章	协议操作者	268
11.1	深入介绍	268
11.1.1	URL 内幕	268
11.1.2	URLConnection 内幕	269
11.1.3	URLConnection 子类	271
11.1.4	协议和内容操作者	272
11.2	快速解决方案	275
11.2.1	获取 URL 的数据	275
11.2.2	获取 URL 的内容	276
11.2.3	设置请求标题	276
11.2.4	读取响应标题	277
11.2.5	使用特定的 HTTP 连接	277
11.2.6	传送数据到服务器	277
11.2.7	打开一个 JAR 文件作为 URL	278
11.2.8	创建一个客户协议操作者	279
11.2.9	安装一个客户协议操作者	280
11.2.10	创建一个客户内容操作者	281
11.2.11	安装一个客户内容操作者	282
第 12 章	解释 HTML	284
12.1	深入介绍	284
12.1.1	显示	284
12.1.2	处理 HTML	286
12.1.3	实现 Ad Hoc	287
12.1.4	Ad Hoc 细节	289
12.1.5	使用 AHParse	290
12.1.6	处理图像	293
12.1.7	属性解析	293
12.1.8	改进可用性	294

12.1.9	再次访问 Swing	296
12.2	快速解决方案	298
12.2.1	通过 Swing 使用 HTML	298
12.2.2	使用 JEditorPane 显示 HTML	298
12.2.3	通过超链接显示 HTML	299
12.2.4	使用 AHParse	299
12.2.5	通过 Swing 解析标签	299
12.2.6	通过 Swing 解析属性	301
12.2.7	通过 Swing 解析文本	301
第 13 章	HTML 服务	302
13.1	深入介绍	302
13.1.1	关于 JSP	302
13.1.2	定制 Tandem 中的服务	308
13.1.3	通过代理创建的 Web	317
13.1.4	拍卖服务器	324
13.2	快速解决方案	337
13.2.1	使用 JSP 进行服务器端编程	337
13.2.2	从 JSP 中读输出	337
13.2.3	在 JSP 中向浏览器写数据	338
13.2.4	使用 JSP 页的定向功能	338
13.2.5	写一个简单的 Web 服务器程序	338
13.2.6	配置 HttpServer (Http 服务器)	339
13.2.7	定制 HttpServer	339
13.2.8	写一个代理服务器	342
13.2.9	调试一个代理服务器	343
第 14 章	XML	346
14.1	深入介绍	346
14.1.1	进入 XML	347
14.1.2	XML 语法	348
14.1.3	有效的 XML	349
14.1.4	文档对象模型 (DOM)	351
14.1.5	名字空间	351
14.1.6	Java 对 XML 的支持	352
14.1.7	XML 库	354
14.1.8	使用 SAX	355

14.1.9	使用 DOM	366
14.2	快速解决方案	372
14.2.1	安装 Java XML 扩展	372
14.2.2	在 JSP 里创建 XML	372
14.2.3	创建一个解析器	374
14.2.4	创建一个验证解析器	374
14.2.5	创建一个理解命名的解析器	374
14.2.6	使用一个 SAX 解析器	375
14.2.7	使用 SAX 进行验证	375
14.2.8	在 XML 文件里创建 DOM	376
14.2.9	读属性	376
14.2.10	构造一个 DOM	377
14.2.11	编写一个 DOM	378
第 15 章	安全性略谈	382
15.1	深入介绍	382
15.1.1	加密技术回顾	383
15.1.2	Java 安全性	385
15.1.3	关于证书	386
15.1.4	隐藏数据	387
15.2	快速解决方案	395
15.2.1	创建一个安全的套接字工厂	395
15.2.2	创建一个安全套接字	395
15.2.3	与一个安全的 Web 服务器相连接	395
15.2.4	使用 Steganography	397
15.2.5	包含证书	398
15.2.6	显示证书	399
15.2.7	导入证书	399
15.2.8	导出证书	399
附录 A	一些有用的 RFC	400
附录 B	端口的分配	430

第 1 章 Internet 基础

1.1 深入介绍

以我现在的年纪，仍然记得很早以前打一个国际长途电话是一件很麻烦的事。那时候你要打长途，不得不请求话务员来帮助你。不仅如此，话务员一般会记下你的电话号码，在电话接通以后回呼你。

一旦话务员与你联系上，你获得的还是充满噪音和回音的低质量的服务。然而真正让人苦恼的是接下来的付款单。今天，国际长途电话很容易打。整个电话网络只是人们不太注意的现代奇迹之一。但是，无论打电话多么容易，还是存在着功能上的问题：那就是语言问题。如果你不能与对方（中国）共享一种语言，那么打电话到中国是没有用的。即使打的是国内电话，你可能还是不能与传真机或者是 modem（调制解调器）直接交谈（除非你比我更疯狂）。

Internet 就像一个电话网。Internet 中的基础网络设施允许任意两台计算机连接。然而，要使一台 IBM 大型机和一台 PalmPilot 相连，必须有多于一个连接。这两台计算机必须遵循共同的交谈主题以及数据格式。

计算机允许通过各种各样的协议进行通信。其中的一些协议对你来说应该非常熟悉。例如，HTTP（超文本传输协议）是一种协议，允许 Web 浏览器来获取 Web 页面。在更深层次上，整个 Internet 范围，低层协议控制管理着原始数据流。

1.1.1 Java 怎么样

Java 对 Internet 协议来讲，特别重要。为什么？因为至少在理论上，相同的程序可以在不同类型的计算机上运行。如果你正连接一台 PC（个人计算机）至一台 Unix 工作站，事情将变得更简单易行，如果它们都运行完全相同的程序。

Java 的“写好一次，处处能运行”的思想比较适合 Internet 编程。并且，标准的 Java 库有很多实用的类，它能摆脱传统的网络编程的痛苦。

使用 Java 类库，构造一个与服务器的连接很简单，只要声明一个新的 Socket 对象就可以了。你将需要知道服务器方的地址（就像电话号码）和一个端口号（一个扩展）。建立一个用于监听的服务器同样简单。

协议与 Java 一样，对 Internet 上的处理很重要，但协议不是专门针对 Java 的。因此在本章剩下的部分里，不要担心 Java。在第 2 章里你会看到 Java 与网络存在着更多的关系。

1.1.2 协议包

Internet 中有惊人数量的协议在使用中。有很多协议是专用的，你可能从来不会使用它。而对于普通的协议来讲，很多都是相互支持的，使得生活更加方便。

例如，看看 Telnet（远程登录）协议。你可能已经用过 Telnet 程序来登录远程计算机。当你做这件事情的时候，你先要确认三件事。首先，在你的机器上要使用 Telnet 客户端程序。接着你要登录的计算机必须有一个 Telnet 服务器（或者是按 Unix 说法，叫做守护进程，也称作“精灵”）。最后，客户端与服务器端就可以使用 Telnet 协议进行通信。

这样，Telnet 客户端使用 Telnet 协议来与 Telnet 服务器端通信。看起来并不令人惊奇，然而 email 客户端实质上是使用相同的 Telnet 连接与 SMTP（简单邮件传输协议）服务器端进行对话的。邮件消息使用与众不同的方式来表示数据，Web 服务器也使用与邮件消息相同的格式。同时，Web 服务器也使用类似于 Telnet 的连接。

因此，尽管学习这么多的协议可能看起来令人沮丧，事实上，很多高层协议是建立在低层协议的基础上的，这样会使得学习曲线不会像想象中的那么陡。当你要开发更复杂的协议应用时，你可以重复利用你所知道的更简单的协议。

另一件可以使你的生活更加方便的事是大量的源码可以从网上获得，允许 Java 程序使用不同的协议工作。很多与你能想象得到的任何协议相关的开放源码包和例子，在网上都有。Java 本身还有添加常规协议句柄的内在支持功能。同时，Java 的面向对象方法使得用它来创建处理 Internet 协议的复用块显得很自然。

尤其让人感兴趣的是来源于 Jakarta 工程 (<http://jakarta.apache.org>) 的网络组件包。Jakarta 工程是开发出的流行 Apache 网络服务器的一个 Java 小组。NetComponents（网络组件，由 David Savarese 提出）包含着你在网上碰到的常用协议的类。

Jakarta 是一个查找有用的 Java 类的好地方，但并不都是 Internet 相关类。另外一个有趣的工程是 Giant Java Tree (www.gjt.org)。另外，你将会在网上别的地方以及这本书里发现大量的源代码。

1.1.3 Internet 地址

如果将 Internet 视作一个电话网，就需要知道怎样去“呼叫”不同类型的计算机。实际上有几种不同的方法来选择你需要使用的具体的程序。假设你在给银行打电话，就一定要知道银行的主机号码。当你呼叫的时候，你可能进入自动系统，不得不进入需要的部门的分机，如借贷部门。当然，借贷部门的电话可能不间断地服务，以让他们能在同一时间为很多呼叫者服务。

Internet 上的计算机存在着相同的情况。每台机器都有一个 IP（网际协议）地址，看似被圆点分隔开的 4 个 0 到 255 之间的 10 进制数（例如，192.16.32.182）。每个数都是一个 8 位组，因为它代表着 8 个二进制位。IP 地址就与银行的主机号码相对应。

当然，一台机器可以提供很多服务，包括电子邮件、Web 文档、文件传输，以及其他服务。你需要一定量的扩充。这就是端口号。1023 及其以下的端口号保留用于一些众所周知的服务。例如，Web 服务常使用 80 端口。那样，任何 Web 浏览器都能通过连接服务器在 80 端口请求来获得 Web 页面。在快速解决方案一节中你会找到一个端口号的清单。

正如借贷部门相同的分机上有多条接线，相应的，服务器可以在同一个端口上响应多个请求。因此，许多 Web 浏览器可以立刻访问到服务器。当然，正如小公司可能只有一条电话线一样，一个服务器可以选择同一时段只处理一个请求。这要由这个服务器的使用者来决定。

计算机如何获取 IP 地址呢？在某一级别上，一个中央的授权机构 ICANN（Internet 名字与数字授予协会）分配 IP 地址的组织块。但对大多数人而言，他们计算机的 IP 地址是由他们的 Internet 服务提供商或者是网络管理员提供的。对客户机而言，使用动态主机配置协议（DHCP）从一系列可用地址中动态分配 IP 地址是很普遍的。而对于服务器而言，这样做不是个好办法，因为客户机可能在所有的时间里都依赖于这个统一 IP 的服务器。

IP 地址的值实际上是有意义的，并不是随意指定的。它们分为三大类。每大类使用不同数量的位来表示网络号。任何使用相同网络号的机器都在同一个网络里。发向其他网络的请求必须通过路由，以到达目标网络。

很少有人使用 A 类地址，因为它总共只有 126 个。在 A 类地址里，第一个 8 位组指定网络号，即首 8 位相同必定在同一个局域网里边。余下 3 个 8 位组用于指定主机地址。这意味着每一个 A 类网可以容纳 1,600 万台计算机。B 类地址的前两个 8 位组表示网络号。它允许每个网络拥有 64,000 个不同的地址。最后，C 类地址使用前 3 个 8 位组表示网络号，每个网络只拥有 254 个 IP 地址（有些地址保留用于广播）。

在一个新的计划里，不分类别的域间路由或者 CIDR 使用不同数量的位来指定网络号。例如，你将会听到 24 位地址，意味着使用头 3 个 8 位组表示网络地址（与 C 类地址相同）。

ICANN 保留了几个地址段用于局域网测试。例如，这些地址可以用于对自己的专用网进行排错。A 类地址以 10 开始，B 类地址从 172.16 到 172.31，C 类地址从 192.168 开始都保留给了私有网络。在这些地址段的计算机不能直接与 Internet 相连，如果试图连接，路由器和其他 Internet 硬件将忽略它们。

提示：另一种专用地址段的起始地址为 127。这些地址用于指向自身机器（通常是 127.0.0.1）。你可以经常使用这个地址来连接本机而不管它的真实 IP 地址。很多计算机都能认识这个名字 localhost，并使用上述地址作为 localhost 的地址。

1.1.4 DNS: Internet 上的电话簿

长电话号码比较难记，起码它与记住像 192.48.12.101 这样的 IP 地址一样难记。为避免混淆，Internet 支持分等级的服务器系统就是通常所说的域名服务器（DNS: Domain Name Server）。

当访问一台服务器如 www.coriolis.com 的时候，你的计算机要查询一个本地数据库，通

常称为主机文件或者主机数据库。一些计算机甚至没有这个数据库，这个数据库通常很小。如果服务器的名字出现在这个数据库中，那么你的计算机就使用与之相关联的 IP 地址来查找服务器。最有可能的是，你的机器接着要通过 DNS 服务器（由你的公司或者 Internet 服务提供商提供）进行查询。如果 DNS 服务器找不到服务器的名字，它就向它的上一级 DNS 服务器查询。直到其中一个服务器知道待查服务器的 IP 地址，或者也可以说 DNS 服务器是一簇根服务器（就是说，它没有上一级服务器）。最后，域名服务器向你的机器返回一个 IP 地址或者是错误信息。

DNS 就像一个电话簿。通常它只提供主机号码，而没有提供你要的分机号码（端口号）。另外，只有出现在 DNS 系统中的计算机才拥有可用的名字。例如，如果你拨进了一个 Internet 提供商如 AOL 或者 MSN，你的 IP 地址是从一个集合中随机选出来的（DHCP）。尽管你的机器可能有一个名字，但是 DNS 还是不知道，因此其他机器无法通过机器名来访问你的计算机，然而它们如果知道你机器的当前 IP 地址，就可以访问了。

在 DNS 系统里是不会出现像 127.0.0.1（环回地址）这样的 IP 地址的。然而，很多本地 IP 数据库拥有这个特殊地址（本机地址）的入口。甚至本地机器也可以充当主域名服务器。例如，Windows 2000 就提供了 DNS 服务，用于缓存你查询机器名的入口。这就加快了对同一主机的多个连接。因为本地 DNS 服务器如果遇到不能确认的主机，它就要查询正规的 DNS 服务器，这个过程对于用户和程序员来说是透明的。

1.1.5 URL, URI, URN

无论谁浏览过网页，都要用到 URL（统一资源定位器）。但是，日常网络应用中也有几个不是很明显的地方。

下面是一个典型的 URL，有几个部分你可能不太熟悉：

`http://aaw:startrek@www.coriolis.com/jbb/go.jsp?v1=100&v2=doctor`

考虑 URL 的每一个部分：

- `http://`——这是协议标识符。在这里，协议是 HTTP（超文本传输协议），通常是为了获取网页。而一个安全的网页会使用 `https://`，而一个 FTP 地址会将使用 `ftp://`。
- `aaw`——这是一个用户的 ID，用于登录并访问这个资源。当然，很多 URL 不要求登录，因此这个是可选的。
- `startrek`——如果你提供了一个用户的 ID，你就可能要同时提供密码。当然，这个密码是未加密的。你可以将密码置空，让浏览使用一个你已经告诉它的密码或者是它提示你输入密码。
- `@`——如果输入了用户名（或者用户名和密码），就要使用 @ 符号来标志主机名字的开头。没有提供用户名，就不用 @ 标识。
- `www.coriolis.com`——将浏览器指向本地机器的主机名。这个地址可以是 IP 地址，也可是域名（这里用的就是域名）。

- `/jbb/go.jsp`——它在 IP 地址里是待请求文档的路径。在没有指定文档名（或者只指定路径名）的情况下，许多服务器将提供一个缺少的文档给你。
- `?v1=100&v2=doctor`——这部分是可选的，是一个查询串。客户端用这个方法可以向服务器方发送数据。例如，一些 Internet 表单利用查询串向服务器方提交它们的数据。

协议标识还要选择客户端方连向服务器端的端口。主机名是服务器的标识。客户端向服务器端发送其他信息，但不一定对其内容感兴趣。

下面是另一个 URL：

`http://www.al-williams.com/glossary.htm#solder`

符号 `#` 终止了文档名，并且提供一个哈希串。对一个网页而言，它表示浏览器载入文档以后重新定位的锚（anchor）。

URL 与原始地址和数据比较而言，容易处理一些。但 URL 的最大优点在于它为用户提供了一种协议无关的方式来访问 Internet 资源。例如，如果你在 Web 浏览器里输入 `telnet://www.coriolis.com`，浏览器将意识到无法联系使用 Telnet；因此，它会启动一个外部的 Telnet 程序来完成这件工作。

你也许偶尔听说过缩写词 URN（统一资源名字）和 URI（统一资源标识）。通常你可以认为，它们可以与 URL 相替换。URI 是简单的串，用于独一无二的表征任何资源，而不一定是 Internet 资源。例如，这本书有一个 ISBN（国际标准书号），就可以把它看作是这本书的 URI。很明显，ISBN 不是 URL，而是 URI。

URN 也是一种 URI（由 RFC2141 中的定义部分可以看出）。URN 拥有持久属性以及本地独立属性。例如，假设你的网站要移到不同的服务器上，但你只想让用户通过单个 URL 来访问你的网站，你只需要在 `http://purl.oclc.org/` 上创建一个 URN（在这种情况下是一个持久 URL（Persistent URL）或简称 PURL），以提供一个持久的 URN，将一直指向你网站的当前 URL。

1.1.6 层（Layers）

现代的网络使用层的概念，有时也被称作协议栈。可以这么认为，每个层都依赖于所有比它更低的层，这就是为什么称作栈的原因。典型的开放系统互联（OSI）模型的栈是七层结构。但是，对于 Internet 而言，逻辑上可以将其分作四层：应用层，传输控制协议（TCP）层，IP 层，以及物理接口层。

在栈的底部是物理接口层。它是一种软件，用于驱动网卡、调制解调器或者其他任何与网络互联的设备。在物理层的上部是 IP 层。IP 层处理原始的地址与数据信息包（也称作数据报）。

数据报有个缺点是不能确保数据报到达目的地。接收报文的顺序也许不同于它们发送时的顺序。有关终点计算机的网络错误或者问题可能觉察不到。同时数据报也有优点，就是它的效率比较高。

要解决数据报的这种固有问题，要依赖于协议栈的上一层 TCP 层。TCP 实际上管理着 IP 数据报带来的不确定性。当你使用 TCP 发送数据时，TCP 层期待着接收目标计算机的确认信

息。如果没有确认，TCP 层最终将重发数据报。

TCP 层不仅确保每个数据包的到达，而且要将收到的所有数据包以正确的顺序汇集在一起，这样接收者将得到与发送时顺序一样的数据包。可以想象，就原始性能而言，TCP 不如 IP 层高效。但是，当计算机间需要准确的逻辑数据流时，TCP 是好方法。某些程序要求很高的性能，并且使用自己的方法检测丢失的或者错序的数据包，就可以使用 TCP 的连接方式。

协议栈的顶层是应用层。这一层是本书集中要讨论的一些协议。HTTP, FTP 和 SMTP 都是应用层协议的例子。

传统的网络方面的书，都指出协议栈的每一层只与和它相邻的上一层和下一层发生联系。例如，应用层不会与物理网络硬件发生联系。如果要为计算机写一个网络协议栈，这就很重要了。但是，对常规的网络工具而言，重要的是：逻辑上每一层只与另一计算机的对应层通信。不要认为那是错误的方式。它并不表示该层与远处的对应层直接相连。但可以假设是这样的，例如，可以假设 TCP 层与另一端机器的 TCP 层是直接相连的，即是对等层。当然，真实情况不是这样的。打开一个 TCP 连接有可能通过拨调制解调器产生 X.25 数据包，然后发射数据到地球卫星上。但你并不关心，错误地认为 TCP 套接字与对等层相连。

1.1.7 基本协议

什么是数据报呢？每个数据报都有一个短的头和填载数据。每个数据报都有一个最大尺寸，它依赖于网络。当前 Internet 使用的是 IPv4 的版本，但是随着 IP 地址需求的快速增长，使用 IPv6 来取代 IPv4 是一个趋势，它有不同的格式，并且允许更长的地址（JDK1.4 中就增加了对 IPv6 的支持，JDK1.4 是 Java 开发工具集的一个版本）。

数据报的第一部分包含着版本号，目前缺省值总是 4。据推测，将来网络硬件将用这个值来区分版本 6 和版本 4。

数据报的下一部分指出报头的大小（最大 60 字节）。也有一个域值表示整个数据报的长度。因为一个数据报可能不止一次到达（由于网络路由的缘故），每个数据报也可以有 ID 号来帮助识别重复的数据报。

数据报的其他域有的表示协议类别（由 ICANN 分配的协议号），有的表示源 IP 地址和目的地 IP 地址，还有的域表示报头的校验和。大量的位用于表示一些选项，其中一些并不常用。另外报头有一个域（你可能听说过）叫做存活时间段（TTL）域。数据包的 TTL 值决定着包在系统抛弃它之前可以重发多少次。当数据报在计算机、路由、网关、中继器或者其他网络硬件间传输时这个域时可以帮助阻止无限的循环。每一跳段将 TTL 的值减 1，直到 TTL 值减为 0。

TCP 并不高效，但是可靠。然而，有些应用需要尽可能快的速度。例如，在视频流的应用中，丢失一些数据报相对于获取高速度来说是可以忽略的。因为每一层只与相邻的上下层对话，TCP 允许你创建用户数据报（UDP）套接字。这些套接字与原始 IP 套接字非常接近。它们不太可靠，但是很高效。

除了 TCP 和 UDP, IP 层上边还有其他协议。例如, 当你使用 ping 程序来检验网上一个主机是否可达时, 你就在使用 ICMP (Internet 控制消息协议)。然而, Java 在 JDK1.4 以前的版本里边不允许你直接访问 IP, 也不支持除了 TCP 和 UDP 以外的套接字。如果你要用 Java 写一个 ping 程序, 你就得借助于另一种语言 (如 C 语言) 写这段代码, 并将这段代码加到你的 Java 程序中去 (或者用 JDK1.4 版的 Java 编写程序)。

1.1.8 网络硬件

逻辑上, 当连接浏览器和远程服务器时, 就有一个从你的 PC 和服务器的直接连接。但在实际生活中, 连接路径比你猜想的要复杂得多。

计算机可能要通过各种网关计算机甚至是路由器连向 Internet。并且, Internet 提供商多半使用了交错的路由、交换机和其他秘密硬件来与 Internet 骨干网相连。你想要连接的服务器很可能也要经过这么多数量的硬件。

不过你不用担心。除了降低数据传输速度以外, 这些路由器、网关、交换机是不可见的。

Internet 是一个“双向街道”。当连向 Internet 的时候, 你开放了一部分信息。但是, 你同时也将自己的计算机暴露出来, 易受到潜在的窥视和彻底的攻击。当有一个固定的 IP 地址时尤其如此。事实上, 即使你用动态分配的 IP 地址, 照样会受到攻击, 但如果有人专门找你, 也会很费时间的。

1.1.8.1 防火墙

为了保护自己, 很多公司甚至是个人用户都在自己的计算机和公用 Internet 上安装了防火墙。防火墙可以是专门的一片硬件, 或者是运行在一台计算机上的软件。

当局域网上的计算机试图与外网交互时, 防火墙将截取数据报。它将根据个人在防火墙上设置的一套规则来传递或者删除数据报。这些规则可以过滤数据报报头上的任何内容。例如有这样的规则, 你可以允许来自固定的 IP 地址的 FTP 请求有效。一些协议 (如 Telnet) 可能完全关闭。一些防火墙也可以执行网络地址转换 (NAT) 将一个私有网络连到公用 Internet 网上。这样一个只有一个 C 类地址的站点就可以将几十台计算机连到 Internet 上。

1.1.8.2 代理服务器

防火墙工作在网络协议栈的底层, 因此它不会干挠 Java 程序, 除非遇到一个攻击你的程序。但是有一种类似的设备代理服务器工作在应用层。这些代理服务器通过网络请求具体的路由, 因此比较熟悉正在使用的协议。

例如, 假设由你来负责一个公司的网络, 而公司需要监控对网站的访问。你可以将 80 端口阻塞 (使用防火墙), 接着在同一台计算机上安装一个代理服务器来操纵防火墙。代理服务器可能使用如 8888 这样的端口。现在办公室中的任何人必须设置他们的代理服务端口为 8888。代理服务器将请求在 8888 端口和 80 端口之间来回传递。

这将允许你登录所有网页访问一些被人们认为是侵犯隐私的内容。你也可以通过将一些 IP 地址列入黑名单来禁止访问。许多代理服务器具备某些附加功能, 依靠对底层协议的知识

而获得。例如，一个网站代理服务器可以将内容缓存（为了更快的重载速度），预先获取链接（为了加速网页浏览），或者禁止访问广告。

尽管 HTTP 代理服务器是最常见的，你也可以有任何别的定义好的代理服务器，如 FTP 代理或者 SMTP 代理。即使有些代理服务器知道多种协议，也没有一个代理服务器能处理任何一种协议。这对程序员来说是一种劣势。如果你的程序想通过代理服务器来执行，最好使用一种代理服务器能处理的协议。

有个标准叫 SOCKS，允许穿过代理服务器，但不是所有的代理都遵循这个标准。如果使用常见的代理协议如 HTTP、FTP，或者有一个 SOCKS 代理，那就可以设置 Java，让它来处理代理。这使得代理服务器不会太影响工作，如果知道怎样正确设置 Java 的属性的话（参考 1.2 节“快速解决方案”）。

1.1.9 协议的学习

Internet 是非常民主的。所有的标准协议都用于建议人们如何使用通信。提出这些标准的主体是 Internet 工程任务小组(IETF)。当有人想为 Internet 定义一种新的标准时，他可以向 IETF 提交一份 Internet 草案。很多文档是用于定义标准的，但也有其他一些只是信息文档。

最后，IETF 可能同意草案，将它制成 RFC（注释请求文档）。所有被 IETF 通过的标准都是 RFC。但是，有些 RFC 不是真正的标准。RFC 一旦发行，就很少改变。大多数改变需要一个新的 RFC。有些协议如电子邮件格式的标准曾经改变了好多次，也就有很多与之相关的 RFC。

只有很少的 RFC 能成为 STD（标准）文档。现在每天使用的很多协议并不在 STD 一级，由于政治上的原因它们永远也不会成为 STD 文档。标准有需求上的级别。一些协议如 IP，是必需的。所有 Internet 上的主机必须实现 IP 协议。有些协议只是推荐使用但不是必需的。即使看起来很基础的协议如 TCP 也是推荐使用的。其他 RFC 分为可选的、有限使用的，或者不推荐使用的。

另一个类似的组织叫 WWW Consortium (W3C)，即 WWW 联盟。这个组织由几个供应商（不是开放团体，如 IETF）来管理，它们维护的标准有 HTML 和 XML（扩展标识语言）。任何人都可以提交 IETF 文档，但只有 W3C 的成员方可提交新的 W3C 标准。

除了标准以外，还可以发现大量的用 Java 编写的与网络相关的开放源码软件。开放源码软件对开发人员来讲是一大法宝。不仅可以获得工作用的软件，还可以获取它的源码。如果你对它本身很有兴趣，甚至可以参加进去，帮助开发该软件。

大量的 Java 软件可以在某些类别的开放源码组织中获得。下面几个地方就可以去看一看。

- Jakarta——Jakarta.apache.org
- Sourceforge——www.sourceforge.net
- Java Boutique——javaboutique.internet.com
- Giant Java Tree——www.gjt.org

- GNU Java Programs——www.gnu.org/software/software.html#Java

1.2 快速解决方案

1.2.1 确定你的真 IP 地址

如果你经常开发程序，就有必要知道你的真 IP 地址。当然，你总是可以使用环回地址（通常是 127.0.0.1）。你的操作系统可能支持 `ipconfig` 或者 `winipconfig` 程序，能告诉你有关你机器里边各种网卡的 IP 地址。

但是，有时候这些结果容易误导你。如果你使用了 NAT 防火墙，你的公共地址将与你机器的地址不匹配。一些 Internet 连接，如卫星和某些电缆系统，也给你一个很难算出的 IP 地址。

一旦出现这些情况，你可以求助于基于 Web 的服务，由它报告你的 IP 地址。在任何一处搜索引擎上搜索“IP Reflector”（IP 映射）就能出现很多结果。其中有一条结果在 www.dslreports.com 的“Tools”那一节工作得很好。

这些服务只是简单地描述了 Web 服务器提示出来的你正在使用的地址。一旦你在使用 NAT 地址转换，这个地址就可能很难算出。

注意：如果使用卫星或者电缆调制解调器（使用常规电话的调制解调器）来传输数据，用特殊的电缆调制解调器来接收数据，许多反射器将报告错误的 IP 地址。那是因为你的系统实际上拥有两个 IP 地址，一个用于传送，一个用于接收。DSLReports 的反射器能够工作，但是其他一些反射器将返回你的传送地址，通常不是很实用。

1.2.2 使用动态重定向

如果你计算机用的是静态 IP 地址，就可以直接访问一个 Web 服务器。这大体上可以用于测试或者向外部世界显示你的工作。然而如果采用拨号连接、DSL、无线或者卫星连接，你的 IP 地址在你连向 Internet 时将发生改变。这就让别人难以发现你，甚至你自己也难以发现自己。

一种解答是使用 IP 重定向服务。在 Web 上有好几种免费的服务。当你连向 Internet 的时候，你可以手工更新你的 IP 地址，或者在某些情况下重定向服务会启动程序自动完成这项任务。

这种类型的服务会在 DNS 系统里给你一个名字，使别人以常用的方式能找到你。然而当浏览器通过名字来访问你的站点时，重定向服务会将名字映射到你最后一次使用的地址。当然，那得假设你的计算机还连着那个地址。有些重定向服务能算出你在不在，并将访问者重定向到替补网页上（就像打电话时如果人不在时自动回复一样）。

这类服务有些是比较流行的，如在 www.cjb.net, www.dns2go.com, www.dyndns.org 里头。

当然，如果你拥有一个向公众开放的站点，就可以在每次你登录时更新一个小的重定向页面，也许是用一个 Java 程序，将用户带你更新过的 IP 地址里去。

最简单的情况，你可以写一个类似于下面的 HTML 文档：

```
<HTML>
  <HEAD>
    <META HTTP-EQUIV=REFRESH CONTENT="http://199.28.1.1;0">
  </HEAD>
  <BODY>
    <H1>Redirecting...</H1>
  </BODY>
</HTML>
```

你可以在每次登录并且知道你的新 IP 地址的时候在<META>标志里修改 IP 地址。Internet 上有很多程序可以自动完成这个处理。在站点 www.webattack.com 上的 IP 地址工具一节里可以看到能获得些什么。

你会发现在 http://dir.yahoo.com/Business_and_Economy/Business_to_Business/Communications_and_Networking/Internet_and_World_Wide_Web/Forwarding/ 上边列出的 DNS 供应商的列表很有用。它既覆盖了免费服务也覆盖了那些收费服务。

1.2.3 确定 IP 地址的类型

如何确定 IP 地址的类型呢？现在来看看。表 1.1 能帮助你确定所使用的 IP 地址的类型。关键是将 IP 地址看作是 32 位二进制数而不是 4 个独立的 8 位二进制组。头两位告诉你地址的类型。在表 1.1 中，最重要的位是第 31 位，第 0 位是最不重要的。

表 1.1 判断 IP 地址的类别

类型	第 1 位 (位 31)	第 2 位 (位 30)	网络地址部分	主机地址部分
A	0	任意值	位 30 到位 24 之间	位 23 到位 0 之间
B	1	0	位 29 到位 16 之间	位 15 到位 0 之间
C	1	1	位 29 到位 8 之间	位 7 到位 0 之间

1.2.4 选择端口号

选择端口号有两种方法。如果你在设计一个服务器，那就要选择一个端口号用来表示服务器的类型，或者表示客户端希望服务器与它之间通信的方式。如果在客户端上工作，就不用注意所使用的端口号，要注意的是服务器的端口号。表 1.2 显示了一些常用的端口分配。记住 TCP 端口和 UDP 端口是独立的，尽管很多服务器用同一个端口号来监听 TCP 和 UDP 端口。

表 1.2 常用端口号

服务名	端口号	协议	相关描述
echo	7	TCP/UDP	测试服务——将数据回送给发送者
daytime	13	TCP/UDP	检索当前时间和日期 (ASCII)
quote	17	TCP/UDP	返回对当天的引用
FTP (data)	20	TCP	文件传输协议 (数据通道)
FTP	21	TCP	FTP 命令
ssh	22	TCP	安全 shell 模式 (是 Telnet 的安全替代方式)
Telnet	23	TCP	登录到远程主机
SMTP	25	TCP	简单邮件传输协议, 发送和提交邮件
time	37	TCP/UDP	以整数格式返回从 1900 年 1 月 1 日算起的时间
DNS	53	TCP/UDP	域名服务器, 将主机名解析成 IP 地址
finger	79	TCP	返回关于一个系统或者用户的信息
HTTP	80	TCP	超文本传输协议 (Web)
POP3	110	TCP	邮局协议 3——读取邮件
NNTP	119	TCP	网络新闻传输协议 (Usenet)
NTP	123	TCP/UDP	网络时间协议

提示: 在 RFC 1700 中你可以找到完整的端口列表。

注意: 在 Unix 以及与 Unix 类似的系统中, 只有超级用户 (root) 可以打开小号端口。那意味着只有超级用户能启动标准的网络服务, 要么它们只能在 SetUID 方式下运行, 在这种方式里, 允许程序假设这个 ID 的用户具备 root 超级用户的权限。

1.2.5 使用 Ping 和其他工具

ping 是最有用的网络工具之一。这个简单的命令允许你判断一台网络上的计算机是否可达。也可以将域名解析成 IP 地址或者将 IP 解析成域名, 还可以显示基本的路由信息。

使用 ping 的最简单的方法就是键入类似如下的一行命令:

```
ping www.coriolis.com
```

它将返回类似于如下的结果:

```
Pinging www.coriolis.com [38.187.128.10] with 32 bytes of data:
```

```
Reply from 38.187.128.10: bytes=32 time=828ms TTL=119
```

```
Reply from 38.187.128.10: bytes=32 time=813ms TTL=119
```

```
Reply from 38.187.128.10: bytes=32 time=703ms TTL=119
Reply from 38.187.128.10: bytes=32 time=781ms TTL=119
```

```
Ping statistics for 38.187.128.10:
```

```
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
```

```
Approximate round trip times in milli-seconds:
```

```
Minimum = 703ms, Maximum = 828ms, Average = 781ms
```

可以看到 ping 将主机名解析成正确的 IP 地址并将它显示出来。

但是，如果你 ping 一个 IP 地址时，会出现什么情况呢？如下所示：

```
ping 38.187.128.10
```

在这种情况下，ping 不会将 IP 地址解析成主机名。但是，如果你增加一个选项 -a，那么 ping 命令将执行 DNS 反向查询，并显示像第一个例子中出现的名字。

另一个技巧是 ping 可以显示出发出的包怎么从你的计算机到达远程的目的地计算机。简单地加一个 -r 选项和你想记录的跳段数（从 1 到 9）。

另一个功能更强大的跟踪路由的方法是使用 traceroute（或者是 tracert，如果你是 Windows 用户）命令。traceroute 命令的输出显示出数据包所到达的网络的每一部分、到达那个部分的主机所用的时间（程序发三个测试数据包因此显示三个时间值）以及对该路径上每台计算机的解析过的主机名。下面是一个典型的 traceroute 会话：

```
Tracing route to www.coriolis.com [38.187.128.10]
over a maximum of 30 hops:
```

```
 0  282 ms  125 ms  125 ms  twhou-5800-1.ev1.net [216.88.77.2]
 1  109 ms  110 ms  109 ms  216.88.77.1
 2  110 ms  109 ms  125 ms  twhou-7200-1.ev1.net [207.218.245.1]
 3  110 ms  110 ms  125 ms  216.90.223.81
 4  141 ms  141 ms  141 ms  64.242.22.97
 5  234 ms  156 ms  235 ms  204.6.142.50
 6  203 ms  250 ms  219 ms  sw.transit.tier1.us.psi.net [154.13.2.98]
 7  219 ms  500 ms  234 ms  rc10.sw.us.psi.net [38.1.24.202]
 8  219 ms  219 ms  265 ms  salt-lake.psi.net [38.1.44.249]
 9  250 ms  282 ms  453 ms  38.2.190.24
10  313 ms  390 ms  453 ms  www.coriolis.com [38.187.128.10]
```

有时路径上的计算机不会对跟踪进行响应，这时时间值就会显示成星号。同时，如果使用防火墙或者代理，那就可能得不到跟踪值。traceroute 程序在每个跳段都要发送三个数据报。数据报的 TTL 的设置，会使那一跳段的中继点删除数据报。大多数网络计算机在 TTL 过期时会发送消息到数据报的源发方。因此，traceroute 从途中的每台计算机那儿获得消息并能识别它们。

这些工具在你试图推算某些东西是否在工作时显得很有用。traceroute 命令尤其能显示出你的网络路径中经过的防火墙、路由器和其他设备，而用其他方法是看不出来的。

1.2.6 自己动手练习一个协议

当试图理解一个协议的时候，通常有用的方法是自己动手。那就是，不借助于客户端程序。标准的 Telnet 程序允许你直接访问很多服务器，因为很多服务是建立在 Telnet 类型的协议的基础上的。

例如，假设在没有 Web 浏览器的情况下想获取一个网页。可以 Telnet 到主机那里，并模拟一个 Web 浏览器。你只需简单地将 Telnet 程序的端口改为 80 来替代常规的 Telnet 端口（23 号端口）。

试试下面几步：

启动 Telnet 客户端程序。在 Unix 命令提示符下，输入：

```
telnet www.coriolis.com 80
```

你可以将 `www.coriolis.com` 替换成你要连接的机器名字（不要包括 `http://`）。如果在 Windows 下运行，就可以使用相同的命令（实际上，你必须在 Windows 2000 下运行）。如果在 Windows 98 或者 Windows ME 下，可以启动 Telnet 并使用菜单选择你要的主机名和端口。

一旦 Telnet 客户端与主机相连，输入下列命令：

```
GET http://www.coriolis.com
```

如果 Telnet 客户端没有响应字符，你可能键入了不可见字符（你可以在程序里设置本地回显方式）。当你按回车键时，服务器会输出从 Web 浏览器上能看到的数据。在本书的其他部分（第 7、10、12 章）你可以学到有关这种格式的更多内容，你也许可以看到第一部分包含 HTTP 头，接着是空格线，然后是与缺省网页相应的 HTML 代码。

依靠 Telnet 客户端，你也许不能回卷到页的开头部分，尽管好的客户端程序会让你选择很多行来保存。

实际上 Telnet 在获取网页时并不是很有用。但这是试验任何 TCP 服务器时练习使用文本命令的好方法。有大量的服务器对几乎对用户可读的命令都响应。

1.2.7 查找 RFC

如果你想发掘 Internet 方面更深的细节，就得看 RFC 文档。RFC 文档的主要资源在 www.ietf.org/rfc.html 中。你有可能更喜欢 www.faqs.org/rfcs，它也有 RFC 文档，并且还有很多其他 FAQ 文档。

提示：RFC1000 是一个索引，帮助你找到你所要找的具体的 RFC。

表 1.3 显示了一些比较有意思的 RFC 文档。

一些 RFC 文档实际上很有趣，如 RFC2549（“IP Over Avian Carriers with Quality of Service”，即“基于载波服务质量的 IP”）、RFC2324（“Hyper Text Coffee Pot Control Protocol”，即超文本控制协议）。一些只是信息上的，如 RFC1936（“到底什么是 Internet”）。

表 1.3 有用的 RFC 文档

RFC	标题	相关描述
RFC1700	分配的一些序号	端口号以及分配的其他常量
RFC1122, RFC1123	主机要求	定义了 Internet 主机必须实现什么内容
RFC791, RFC919, RFC922, RFC950	网际协议 (IP)	定义了 IP
RFC768	用户数据报 (UDP)	UDP 标准
RFC792	网际控制消息协议 (ICMP)	像 ping 等一系列程序使用的 ICMP 标准
RFC793	传输控制协议 (TCP)	TCP 标准
RFC821	简单邮件传输协议	SMTP (发送和中继邮件)
RFC822	电子邮件消息格式	ASCII 文本邮件消息的格式
RFC854, RFC855	Telnet 协议	通过 Telnet 进行远程登录的标准
RFC862	Echo 协议	用于测试协议
RFC865	对当天的引用	返回一个对当天的引用 (或者消息)
RFC867	Daytime 协议	定义了 daytime 服务
RFC868	时间 (Time) 协议	定义了时间服务
RFC959	文件传输协议	定义了 FTP 服务
RFC977	网络新闻传输协议	定义了 NNTP, 用于处理 Usenet 消息
RFC1000	RFC 参考指南	是对其他 RFC 文档的参考
RFC1034, RFC1035	域名系统	定义了 DNS 服务
RFC1153	邮件的摘要消息格式	定义了一种方法, 将多种 email 消息格式合并成一种
RFC1288	Finger 协议	定义了 finger 服务
RFC1303	网络时间协议 (第 3 版)	允许两个时钟精确同步的服务
RFC1350	一般文件传输协议	TFTP 服务, 用于计算机间传输文件
RFC1738	统一资源定位器 (URL)	定义了 URL
RFC1808	相对统一资源定位器	定义了相对 URL
RFC1939	POP3 协议 (第 3 版)	允许计算机检索邮件
RFC1945, RFC2068	超文本传输协议	定义了 Web 浏览的基本协议
RFC2045, RFC2046, RFC2047	多目的 Internet 邮件扩展	定义了 MIME 类型, 允许二进制数据的 ASCII 编码于 email 或者 Web 浏览
RFC2141	统一资源名字定义语法 (URN)	URN——类似于 URL 的一种方案
RFC2396	统一资源标识 (URI)	URI——另一种标识和定位资源的方案

1.2.8 为代理服务器设置 Java

如果客户软件用在代理服务器的环境之下，不经过一些额外努力你将访问不到公用 Internet 上。Java 允许你在属性文件里设置属性或者在 Java 虚拟机中使用 **-D** 命令行选项。表 1.4 列出了可以使用的一些选项。

表 1.4 Java 代理相关设置

属性	描述
proxySet	值为 true 或者 false，用于指示 Java 是否总使用代理
proxyHost	代理的主机名或者 IP 地址
proxyPort	用于常用代理的端口号
ftpProxySet	值为 true 或者 false，用于指示 Java 是否要用 FTP 代理
ftpProxyHost	FTP 代理的主机名或者 IP 地址
ftpProxyPort	用于 FTP 代理的端口号
gopherProxySet	值为 true 或者 false，表明 Java 是否要用 gopher 代理（一种获取文档的老方法）
gopherProxyHost	Gopher 代理的主机名或者 IP 地址
gopherProxyPort	用于 gopher 代理的端口号
http.proxySet	值为 true 或者 false，表明 Java 是否要使用 HTTP 代理
http.proxyHost	HTTP 代理的主机名或者 IP 地址
http.proxyPort	用于 HTTP 代理的端口号
socksProxyHost	SOCKS 代理服务器的主机名或者 IP 地址
socksproxyPort	用于 SOCKS 代理的端口号

一些代理服务器在被访问时要求使用密码。因为代理服务器是在应用层工作，你使用它的方法也会不同。例如，一个 HTTP 代理可能要你在你的 HTTP 请求里添加一个用户 ID 和密码（在本书的后边将会谈到这些内容）。

第2章 Java 网络编程

2.1 深入介绍

休斯敦与洛杉矶以及我曾经看到的其他城市有很多联系。特别是，每个人都是开车往各个地方去。当我还是小镇里的一个小孩子的时候，去商店都是步行，去其他地方都是骑自行车。现在在休斯敦，几乎没有一个地方是在步行的范围以内。休斯敦就其本身大小而言，几乎没有公共交通。

当生活在像休斯敦这样的城市里时，慢慢你就会知道你实际上就像生活在汽车里边。作为一个软件开发人员，你可能会联系到你最喜欢的编程工具和语言。毕竟，起码你在键盘上度过的时间不少于在汽车里的时间。

但是，不管你在车里呆多长时间，总有些东西你并不常用。我曾经花了 30 秒钟来搜寻紧急闪光灯。每当白天到来时，我不得不苦苦思索设定时钟的过程。我模糊地记得在备用轮胎周围某个地方有一个燃料阀门开启开关，但是我不知道到底在哪儿。

程序设计语言给你的感受很类似。有些东西你每天都用，它们成了你的另一种特性。而有些东西并不是经常用，你不得不苦苦思索什么时候需要它们。

在用 Java 处理 Internet 协议之前，你还要掌握一些有关 Java 工具包的使用技巧。本书假设你都知道 Java 的基本概念。但是，网络编程技术中有些东西你并不常用，因此这一章将告诉你如何利用 Java 的特征，它对上述类型的编程很重要。

Java 对网络套接字支持得很好，并且使用起来很简单。但不一定有机会联系它们的所有特征。并且，一旦有了套接字，必须以有意义的方式来对数据进行编码和解码。如果要写服务器端程序，会发现线程是很基本的部分，如果不用线程，就无法方便地处理多个客户端。

2.1.1 套接字编程的实质

传统的套接字编程使用 C 语言。但是，Java 提供了很多高级方法来处理套接字，使得编写网络程序更加简单。不利的地方在于很难绕过 Java 的内在支持。例如，Java 套接字支持 UDP 和 TCP 连接。如果 ping 程序里需要一些别的东西，如 ICMP（Internet 消息控制协议），那就不得不求助于原始方法调用（很可能用 C 语言编写）或者使用 JDK1.4 或更高的版本。

毫无疑问，Java 对网络的支持，体现在 `java.net` 包里。这个包的很多类在通常的应用中并没有用到。有可能用到的类如下：

- `DatagramPacket`——通过 UDP 套接字（由 `DatagramSocket` 来实现）来发送或者接收

一个数据包。

- DatagramSocket——通过 UDP 通信的套接字。
- HttpURLConnection——使用 HTTP 服务器来通信的类。
- InetAddress——通过名字或者数字来表示 IP 地址。
- JarURLConnection——从本地文件、Web 服务器或者 FTP 服务器使用 JAR 文件的类。
- MulticastSocket——一种用于多点传送的套接字（也就是说，向多个远程套接字发送或者接收数据）。
- ServerSocket——一种用于监听来自客户端的连接套接字。
- Socket——通用套接字。
- URL——表示一个 URL 地址的类。
- URLDecoder——将数据解码成 URL 的格式。
- URLEncoder——对 URL 数据进行编码。

还有一些基本类将在后边几章中描述。这一章里的一些基本的类用于直接处理套接字。

套接字通信的基本思想比较简单。客户端建立一个到服务器端的连接。一旦连接建立了，客户端就可以往套接字里写数据，并向服务器发送数据。反过来，服务器发送客户端要从套接字里读的数据。几乎就那样简单，也许细节会复杂些，但是基本思想就那么简单。

Java 提供了三大类套接字类。DatagramSocket 这个类实现 UDP 协议。回顾第 1 章，UDP

套接字并不使用连接，也不确保数据的传送，不保持数据的顺序。流入和流出的数据都封装在 DatagramSocket 对象里。

另外两个套接字类是 Socket 和 ServerSocket，它们都支持 TCP 连接。如果要连接一个服务器，就要用到 Socket。如果写一个服务器端程序，就要使用类 ServerSocket。为什么不同呢？客户端套接字实际上并不关心它在本地使用什么端口。它确实需要连接到另一台计算机的一个确定的端口上。另一方面，服务器端非常关注自己的本地端口赋值（客户端就是通过那个端口找到它的）。服务器端同时还要监听连向自己的各个连接。

对一个到自己的连接的监听并不是一个非常直观的处理。假设一个 Web 服务器在监听 80 端口。当一个客户端连向该服务器，它好像认为服务器在使用 80 端口来与客户端对话，对吗？实际上它不是这样工作的。如果它的确是这样工作的话，同一时刻就只能有一个客户端能连接到服务器了。实质上，网络软件做这样的安排，使得当客户端连向一个端口时，请求转向另一个使用随机端口号的套接字。只要与服务器相连接，客户端是不会真正在意的，同时服务器端的主套接字可以继续自由地监听进入的连接。

2.1.1.1 定址

不管计划使用什么类型的套接字，有必要确定指定套接字地址的方式。也许认为将主机名或者 IP 地址传递给套接字的构造函数就可以，事实并不是那样的。而是用 InetAddress 来表示远程计算机的地址。

InetAddress 没有任何公有构造函数。因此如何获得对象的一个实例呢？可以使用下述三

个静态方法其中的一个来创建一个 `InetAddress` 实例。

- `getLocalHost` 方法返回指向本地计算机的 `InetAddress` 对象。
- `getByName` 方法返回一个指定主机的 `InetAddress` 对象。名字可以代表 IP 地址的字符串或者是真实的主机名字。
- `getAllByName` 方法查找与指定名字相匹配的所有地址。这个方法返回一个数组。

使用这些调用中的任何一个将返回一个 `InetAddress` 对象（或者对象数组，如 `getAllByName`）或者抛出异常 `UnknownHostException`（如果名字不能解析）。通常，要将 `InetAddress` 对象传给套接字的构造函数。然而，也可以使用对象这种方式来将主机名解析成 IP 地址（像 DNS 的接口那种情况）。可以调用实例方法 `getHostName` 和 `getHostAddress` 返回主机名和 IP 地址。也可以使用 `getAddress` 以字节数组而不是字符串的方式返回 IP 地址。

清单 2.1 演示了一个简单的控制台程序，可以用来解析机器名字或者 IP 地址。当用 IP 地址作为命令行的参数时，程序就会工作，但它可能查找也可能不查找对应的名字。如果 Java 不能解析主机名，套接字的名称就只是代表它的 IP 地址的字符串。

清单 2.1 这个程序将主机名转化成等效的 IP 地址

```
import java.net.*;
public class GetIP
{
    public static void main(String [] args) {
        InetAddress address=null;
        if (args.length==0) {
            System.out.println("usage: GetIP host");
            System.exit(1);
        }
        try {
            address=InetAddress.getByName(args[0]);
        }
        catch (UnknownHostException e) {
            System.out.println("I can't find " + args[0]);
            System.exit(2);
        }
        System.out.println(address.getHostName() + " = "
            + address.getHostAddress());
        System.exit(0);
    }
}
```

清单 2.2 演示了如何使用 `getAllByName` 方法。返回用在主机上的一组 `InetAddress` 对象，你可以通过数组迭代来确定所有 `InetAddress` 对象的特征。

清单 2.2 一台机器可能拥有多个 IP 地址，这个程序将显示这些 IP 地址

```
import java.net.*;
public class GetAllIP {
```

```
public static void main(String [] args) throws Exception {
    InetAddress[] addr = InetAddress.getAllByName(args[0]);
    for (int i=0;i<addr.length;i++)
        System.out.println(addr[i]);
    }
}
```

类 `InetAddress` 并不是很复杂，但是当连接另一台计算机使用套接字时可以用它。由于它的构造函数也接受主机名，因此很少需要使用这个类，但是当想要解析自己的地址的时候它就很有用。

2.1.1.2 TCP 客户端

当你想要连向一台服务器时，就要用到类 `Socket`。创建 `Socket` 最简单的方法就是向它的构造函数提供一个主机名（或者 `InetAddress` 对象）和一个端口号。清单 2.3 中的程序演示了一个连向 Web 服务器的简单程序。在命令行里提供主机名或者 IP 地址。程序并不传输任何数据，但它要检查是否有服务器（这里假设是 Web 服务器）在监听 80 端口并与客户端相连。

清单 2.3 通过与 Web 服务器连接来证明这个连接存在，即使没有传输任何数据

```
import java.net.*;
import java.io.*;

public class WebPing {
    public static void main(String[] args) {
        try {
            InetAddress addr;
            Socket sock=new Socket(args[0],80);
            addr=sock.getInetAddress();
            System.out.println("Connected to " + addr);
            sock.close();
        }
        catch (java.io.IOException e) {
            System.out.println("Can't connect to " + args[0]);
            System.out.println(e);
        }
    }
}
```

如果在本地机器里正运行着 Web 服务，可以使用本地主机（localhost）来测试这个程序。输出结果如下所示：

```
Connected to localhost/127.0.0.1
```

注意对 `InetAddress.toString` 的隐含调用（由 `println` 调用）自动输出主机名和 IP 地址。当然，你也可以在得到主机名和 IP 地址之后，用自己的格式来显示它们。

2.1.1.3 TCP 服务器端

最常用的套接字类型就是 TCP 套接字。使用 TCP 套接字的时候，一台计算机充当服务器，另一台计算机充当客户机。编写服务器端程序要用到类 `ServerSocket`。通过调用带有端

口号参数的构造函数来创建一个 `ServerSocket` 对象。如果要编写一个标准的服务器端程序，将要用与那种服务类型相联系的众所周知的端口号。例如，Web 服务器将会使用 80 端口。如果编写的不是标准服务程序，可以选择一个系统中没有使用的端口号（通常大于 1023）。

试一下下面的命令：

```
telnet localhost 8123
```

程序极有可能报告它不能连向那个端口。如果某端口正在使用，就选择另一个端口。现在，看一看清单 2.4 中的程序。它提供了 8123 端口上的服务。这个服务不做任何事情，但是如果运行了这个程序，Telnet 程序就可以连向 8123 端口。

清单 2.4 简单化的服务器端程序，允许在套接字 8123 上连接

```
import java.net.*;
import java.io.*;

public class Techo {
    public static void main(String[] args) {
        try {
            ServerSocket server=new ServerSocket(8123);
            while (true) {
                System.out.println("Listening");
                Socket sock = server.accept();
                InetAddress addr=sock.getInetAddress();
                System.out.println("Connection made to "
                    + addr.getHostName() + " ("
                    + addr.getHostAddress() + ")");
                pause(5000);
                sock.close();
            }
        } catch (IOException e) {
            System.out.println("Exception detected: " + e);
        }
    }

    private static void pause(int ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {}
    }
}
```

`ServerSocket` 的构造函数接收端口号。如果想改变形式，也可以简单地修改代码，使其从一个属性文件或者命令行中接收端口号。一旦有了服务器套接字，就可以调用 `accept` 函数来监听到来的连接。这个调用会阻塞，因此程序将一直挂起，直到一台客户端计算机连接到它。如果不可接受，那就不得不在一个线程里调用这个函数，线程在本章的后面有讨论。

这个服务器端程序不做任何事情，只是在碰到有连接的时候停顿 5 秒钟。如果你试图同时运行两个这样的程序，那么第二个程序会抛出一个异常。在同一时刻只有一个程序监听一个端口。如果连接一个比较忙的主机，一旦服务器端再次调用 `accept` 函数，系统就能完成连接。通过使用不同的 `ServerSocket` 构造函数可以设置一定的队列大小，但是底层系统不一定负责完成你的请求。

如果需要的端口号已经在使用，构造函数将抛出异常。可以使用这个特性来发现你机器中已经在使用的端口号（参考清单 2.5）。

注意：如果你在 Unix 下运行程序，很可能需要以超级用户（root）在预留的端口（小于 1024）上启动服务器程序。

清单 2.5 使用此程序可以扫描出计算机中正在使用的端口

```
import java.net.*;

public class LocalScan {
    public static void main(String [] args) {
        for (int i=1;i<1023;i++) {
            testPort(i);
        }
        System.out.println("Completed");
    }
    private static void testPort(int i) {
        try {
            ServerSocket sock=new ServerSocket(i);
        }
        catch (java.io.IOException e) {
            System.out.println("Port " + i + " in use.");
        }
    }
}
```

2.1.2 I/O（输入/输出）流

连接到远程主机的真正目的是发送和接收数据。Java 允许访问 I/O 流，这个 I/O 流就像执行文件 I/O 时使用的流一样。可以向任何流增加过滤器，将它转换成读文件或者写文件的形式来处理特殊的数据类型。

类 `Socket` 有两个返回值是流的方法：`getInputStream` 和 `getOutputStream`。可以直接使用这些流。它们分别属于类 `java.io.InputStream` 和 `java.io.OutputStream` 的成员函数。但是，你还可以增加一个或者更多的过滤器以特殊的方式来处理数据。最新版本的 Java 支持文本流的 `reader` 和 `writer` 类，你可以使用 `InputStreamReader` 和 `OutputStreamWriter` 将流转化成 `reader` 或者 `writer`。

在套接字中根据需要还可以使用几个与流相关的类，这些类如下所示：

- `BufferedInputStream`——允许将到来的数据缓冲（也可以使用类 `BufferedReader`）。
- `BufferedOutputStream`——为流增加一个输出缓冲（你也可以使用类 `BufferedWriter`）。
- `DataInputStream`——允许以机器之间可移植的方式来读 Java 的基本数据类型。
- `DataOutputStream`——允许以机器之间可移植的方式来写 Java 的基本数据类型。
- `InputStreamReader`——从类 `InputStream` 中创建一个 `reader` 对象。
- `OutputStreamWriter`——从类 `OutputStream` 中创建一个 `writer` 对象。
- `ObjectInputStream`——读实现了串行化（或者具体化）的对象。
- `ObjectOutputStream`——写实现了串行化（或者具体化）的对象。
- `PrintStream`——将文本数据写到 `OutputStream`（与 `System.out` 流类型相同）里。
- `PrintWriter`——写与 `PrintStream` 相对应的对象。
- `PushbackInputStream`——允许将指定数量的输入字符压栈。
- `StreamTokenizer`——在 `Reader` 对象中将输入流按标记分割。

我特别喜爱类 `BufferedReader`，因为它有一个 `readLine` 方法一次读一整行数据。`reader` 和输入流（或者 `writer` 和输出流）之间有什么区别呢？老的流类型只处理字节数据。新的 `reader` 和 `writer` 类处理的是 Unicode 字符（双字节字符）。

看看清单 2.6 中的服务器端程序。这是一个只允许单连接的简单程序。当你连向服务器端的时候，它将显示若干消息并退出。

清单 2.6 这个服务器端程序向单个客户端显示消息

```
import java.net.*;
import java.io.*;

public class BeerServer {
    public static void main(String args[]) throws Exception {
        ServerSocket ssock=new ServerSocket(1234);
        System.out.println("Listening");
        Socket sock=ssock.accept();
        ssock.close(); // no more connects

        PrintStream pstream=new PrintStream(
            sock.getOutputStream());
        for (int i=100;i>=0;i--) {
            pstream.println(i + " bottles of beer on the wall");
        }
        pstream.close();
        sock.close();
    }
}
```

一旦服务器端程序运行，可以使用 Telnet 程序连向服务器的 1234 端口。例如，键入：
`telnet localhost 1234`

注意 `PrintStream` 对象本身并不做什么事。必须将它与流（这里是指套接字发向外边的流）连在一起。尽管 `PrintStream` 用在这个例子里比较方便，但在真正的网络程序里使用它还得认真考虑。`PrintStream` 有个问题是在不同的平台里处理行结束符方式不一样。并且，`PrintStream` 在不同的平台里处理编码的方式也可能不一样，还有个不好就是经常抛出异常。如果要检查错误，还得调用 `checkError` 函数。

当然，服务器端或者客户端除了提供数据以外，也可能要接收数据。清单 2.7 中的程序将上一个服务器端程序作了一点改动，它提示客户端给定一个起始数字。这次也可以通过 `Telnet` 与服务器端连接，但必须输入一个数字（依赖于你的 `Telnet` 程序，也许看不到输入的数字的回显）。再回车程序就和上一个一样了，但它会用上提供的数字。

清单 2.7 客户端可以指定信息来控制服务器的输出

```
import java.net.*;
import java.io.*;

public class BeerServer1 {
    public static void main(String args[]) throws Exception {
        ServerSocket ssock=new ServerSocket(1234);
        System.out.println("Listening");
        Socket sock=ssock.accept();
        ssock.close(); // no more connects

        PrintStream pstream=new PrintStream(
            sock.getOutputStream());

        // ask for count
        pstream.print("count? ");
        BufferedReader input =
            new BufferedReader( new InputStreamReader(
                sock.getInputStream()));

        // read and parse it
        String line = input.readLine();
        pstream.println("");
        int count = Integer.parseInt(line);
        for (int i=count;i>=0;i--) {
            pstream.println(i + " bottles of beer on the wall");
        }
        pstream.close();
        sock.close();
    }
}
```

当服务器的另一边是用户时使用字符串是很方便的，但是当两个 `Java` 程序对话的时候，你可以使用数据流或者对象流。清单 2.8 是一个返回 `pi` 值的服务器端程序。但是这个值不是

以文本的形式提供的，而是使用了 `DataOutputStream`。

清单 2.8 一个特定的 Java 服务端程序能用专门的流来传送键入的数据或者对象

```
import java.net.*;
import java.io.*;

public class DataServer {
    public static void main(String args[]) throws Exception {
        ServerSocket ssock=new ServerSocket(1234);
        while (true) {
            System.out.println("Listening");
            Socket sock=ssock.accept();

            DataOutputStream dstream=new DataOutputStream(
                sock.getOutputStream());
            dstream.writeFloat(3.14159265f);
            dstream.close();
            sock.close();
        }
    }
}
```

当然，数据流只处理基本的数据类型。如果想要传递整个对象，就需要使用 `ObjectInputStream` 类和 `ObjectOutputStream` 类。这些流类型只使用实现了串行化或者具体化接口的对象。清单 2.9 演示了一个简单的客户端程序如何取得 π 值。这个方法对 Java 程序之间传递值很有用。

清单 2.9 这个客户端程序从清单 2.8 中的数据服务器程序取得浮点型数据

```
import java.net.*;
import java.io.*;

public class DataClient {
    public static void main(String[] args) throws Exception {
        Socket sock=new Socket(args[0],1234);
        DataInputStream dis=new DataInputStream(
            sock.getInputStream());
        float f=dis.readFloat();
        System.out.println("PI=" + f);
        dis.close();
        sock.close();
    }
}
```

这个串行化接口没有任何方法；只是标志一个愿意将自己串行化（也就是说以持久的方式来存储自己）的类。有些类对于串行化处理要求更多的控制，这些类还必须事先具体化 `Externalizable` 接口（它有两个成员）。类通过重载函数 `readObject` 和 `writeObject` 也可以实现

串行化。

在清单 2.10 和清单 2.11 中将会发现服务器端和客户端程序的例子。在这些例子当中，服务器端将一个对象 Hashtable 串行化，但是可以很容易地串行化多个对象并在网络上将它们重构。

清单 2.10 使用 ObjectOutputStream 类可以为所有的对象服务

```
import java.net.*;
import java.io.*;
import java.util.*;

public class ObjServer {
    public static void main(String args[]) throws Exception {
        ServerSocket ssock=new ServerSocket(1234);
        Hashtable hash = new Hashtable();
        hash.put("Dog", "Madison");
        hash.put("Cat", "Lacey");

        while (true ) {
            System.out.println("Listening");
            Socket sock=ssock.accept();

            ObjectOutputStream ostream=new ObjectOutputStream(
                sock.getOutputStream());
            ostream.writeObject(hash);
            ostream.close();
            sock.close();
        }
    }
}
```

清单 2.11 一个特定的 Java 客户端程序，可以从 Java 服务器端读对象

```
import java.net.*;
import java.io.*;
import java.util.*;

public class ObjClient {
    public static void main(String[] args) throws Exception {
        Socket sock=new Socket(args[0],1234);
        ObjectInputStream ois=new ObjectInputStream(
            sock.getInputStream());
        Hashtable hash=(Hashtable)ois.readObject();
        System.out.println(hash);
        ois.close();
        sock.close();
    }
}
```

注意 Java 的 I/O 类是如何互相调用的。从 `InputStream` 类开始, 可以加一个 `InputStreamReader` 层。接着可以增加更多的层来做进一步的处理。例如, 可以使用 `java.util.zip` 中的类来压缩数据 (当然在另一边对数据解压)。包 `javax.crypto` 中的方法允许对安全的消息进行加密/解密。

2.1.3 高级套接字方法

如果上一节使你相信使用套接字不是很难, 那就对了。当然, 如果你喜欢可以让程序变得更复杂一些。但是, 相对于你的绝大多数程序而言, 套接字使用起来是比较简单的。

简化上述几个程序的一个方法是允许在 `main` 函数中抛出异常。在这种情况下, 不必将每个套接字调用都放到一个 `try` 程序块中。当然, 在实际编程中, 你需要注意这些异常并采取适当的行动。

我确实捕获了由 `InetAddress` 引起的异常。然而, `Socket` 和 `SocketServer` 的很多方法都会抛出 I/O 异常 (`IOException`)。有些方法还能抛出套接字异常 (`SocketException`)。

下面是 `SocketException` 的一些常见子类:

- `BindException`——发生在试图在一个已经在使用的端口上创建套接字时。
- `ConnectException`——发生在试图连接一个套接字, 连接失败时。
- `NoRouteToHostException`——指出访问的 IP 地址是不可到达的。

2.1.3.1 套接字选项

在普通的套接字对象上可以设置若干选项。`setSoTimeout` 函数设置 `read` 函数等待数据的最长时间 (单位是毫秒), 在预定时间内不能完成的读操作将会抛出一个 `InterruptedIOException` 异常。缺省设置值为 0, 即读函数一直等待直到完成。`ServerSocket` 中也有一个类似的函数调用是 `setSoTimeout`, 用来设置 `accept` 的等待时间, 同样抛出 `InterruptedIOException` 异常。

尽管通常没有必要, 仍然建议调用 `setSendBufferSize` 和 `setReceiveBufferSize` 为套接字设置缓冲大小。这样会让你在一定的系统下提高程序性能, 不过要依赖于所使用的平台和应用程序的具体行为。

谈到性能, Java 力图将通过套接字发送的数据量减到最少。如果想要数据流越快越好, 那将对性能产生负面影响, 可以调用 `setTcpNoDelay` 函数。如果将参数值设为 `false` 或者不设置, 系统将使用 Nagle 算法 (详见 RFC896) 来优化数据传输。那意味着系统在发送数据包之前试图积累小的数据块。一直积累到远程系统确认了第一个数据包的时候。然而, 对于交互式程序而言, 这也许会让网络连接看起来很迟缓。

由于使用缓冲, 在所有通过套接字发送的数据被传输完之前关闭套接字是可能的。如果还有问题, 可以调用 `setSoLinger`。这个调用指定了在 `close` 函数返回之前试图清除所有未发送数据之前等待的时间, 单位为秒。

可以通过调用 `setKeepAlive` (参数值为 `true`) 设置套接字, 来定期交换那些活的数据包。活的数据包是指在空闲时计算机之间交换的少量无意义的信息。如果你想很快知道一台计算机是否在没有关闭套接字就已经崩溃, 这些数据包就派上用场了。

通常，不再需要套接字的时候，应该调用 `close` 函数。但是，也可以调用 `shutdownInput` 或者 `shutdownOutput` 函数来关闭半个套接字，而不是完全关闭它。

有可能某些套接字的实现不支持某些套接字选项。这就是为什么当调用像 `setTcpNoDelay` 这样的函数时可能要处理套接字异常（`SocketException`）的原因。

2.1.3.2 套接字的构造函数

例程中使用的构造函数就是通常使用的那种形式。服务器端程序需要指定端口。客户端需要指定主机名，作为一个串或者是 `InetAddress`，同时要指定端口。但是，如果你需要，两个类都有复杂的构造函数可用。

类 `ServerSocket` 有个构造函数，带 `backlog` 参数（有值）和绑定地址参数（可选）。Backlog 允许为到来的连接指定多大的队列。这个队列在服务器忙于处理另一个请求的时候，容纳客户端的连接请求。但是，系统有队列大小的上限值，因此使用这个参数时要认真考虑。

绑定地址允许指定服务器程序用于连接的 IP 地址。当单机拥有不止一个 IP 地址（或者因为它有多个网络适配器或者是单个适配器的地址不止一个）时，绑定地址就很有用了。

常规的套接字提供额外的构造函数，让你指定打开套接字时要用的本地地址和本地端口。当机器的 IP 地址不止一个时这是很有用的。

2.1.3.3 使用 UDP

一旦习惯使用 TCP 套接字，UDP 套接字（`DatagramSocket` 的形式）看起来就困难了。要指定的是套接字使用的本地端口，而不是指定连向一个特定的机器。如果没有指定端口，系统将赋给一个随机值。你只能通过 `DatagramSocket` 来发送和接收字节数组。

`DatagramSocket` 中放入的是你想发送的数据。数据包不仅包含数据，还有目的地地址和端口。将这个包作为参数传给 `DatagramSocket.send` 方法。可以调用 `DatagramSocket.receive` 来填充这个包。

如果与一个服务器有多个事务处理，`DatagramSocket` 对象调用 `connect` 是一个好方法。这可以节省时间，因为任何安全检查和开销只发生一次。如果要和其他服务器对话，就需要调用 `disconnect` 函数。

像 TCP 套接字一样，`DatagramSocket` 允许调用 `setSoTimeout` 来设置读操作的最长等待时间。也可以调用 `setSendBufferSize` 和 `setReceiveBufferSize` 来为底层操作系统设置缓冲的大小。

因为 UDP 套接字不使用连接，不能只是对到来的请求作简单的响应。如果要对服务器作应答，可以提取发送方的地址和端口号（使用 `getAddress` 和 `getPort`），组成另一个包发送出去。

为何使用 UDP？因为它高效。同时，一次可以向多个套接字发送。看看清单 2.12，如果你在命令行中带参数 `r` 运行该程序，它将在端口 1111 上监听 UDP 数据包。如果带参数 `w` 后接一个 IP 地址和一句话，运行此程序会将这句话发送到主机的 UDP 的 1111 端口上。在此情况下，程序使用随机赋给的本地端口号。

可以试图让程序使用自己服务器的 IP 地址。但是，如果在不同的机器上运行相同的接收

程序时，可以使用特殊的 IP 地址来广播。要计算这个 IP 地址是多少，就得看看网络号，确定它属于哪一类 IP（参看第 1 章）。也可以看看网络掩码（使用 `ipconfig` 命令）。具体方法是将非网络号的 8 位组替换成 255。因此一个 A 类广播地址可能是 10.255.255.255。看看网络掩码，它也可以告诉你这个 IP 值。例如，假设 IP 地址为 169.254.39.44，网络掩码为 255.255.0.0。那么该网络的广播地址就是 169.254.255.255。

当向广播地址发送数据时，网络上所有的 UDP 监听者都会潜在地收到一份数据的拷贝。“潜在的”在这里是关键，不要忘记 UDP 数据的投送是无保证的。

清单 2.12 可以使用两份该程序来试验 UDP 套接字

```
import java.net.*;

public class UDPO {
    // command line arguments:
    // r -- read an incoming packet
    // w hostname word -- write word to hostname
    public static void main(String[] args) throws Exception {
        byte [] ary= new byte[128];
        DatagramPacket pack=new DatagramPacket(ary,128);
        if (args[0].charAt(0)=='r') {
            // read
            DatagramSocket sock=new DatagramSocket(1111);
            sock.receive(pack);
            String word=new String(pack.getData());
            System.out.println("From: " + pack.getAddress()
                + " Port: " + pack.getPort());
            System.out.println(word);
            sock.close();
        } else { // write
            DatagramSocket sock=new DatagramSocket();
            pack.setAddress(InetAddress.getByName(args[1]));
            pack.setData(args[2].getBytes());
            pack.setPort(1111);
            sock.send(pack);
            sock.close();
        }
    }
}
```

提示：如果没有紧急要求，可以使用 `URL` 类来获取数据，只要 Java 知道你所用的协议。用一个代表 URL 的串可以构造一个 `URL` 对象。如果想从 URL 中获取数据，可以调用函数 `openStream`。也可以调用 `openConnection` 函数得到 `URLConnection` 对象，从而能更多地控制这个处理。在第 10 章会有这些对象的更多介绍。

2.1.4 线程

在写服务器端程序的时候，通常不用让一些客户端等待另一些客户端。处理客户端的最好方法是为每一个客户端创建一个线程。Java 有内置的线程处理，因而处理起来相对比较简单。

构造线程有两种选择。第一，可以扩张线程对象形成新的对象，来创建线程。如果这个不行，也可以让你声明的对象实现 `Runnable` 接口和相应的 `run` 方法。如果使用第一种方法，启动线程可以使用 `start` 函数（不要直接调用 `run` 函数）。然而，如果要使用第二种方法（实现 `Runnable` 接口），就得创建一个线程对象，并将你的对象作为参数传给线程的构造函数。

在清单 2.13 里会发现使用线程的服务器应用的例子。注意 `MTThread` 对象扩展了线程类。`main` 函数还是创建了 `ServerSocket`，并调用了 `accept`。但是，当客户端连接时，`main` 函数创建了 `MTThread` 的一个新的实例，也就是新的线程。新的线程需要知道客户端的套接字，因此 `main` 函数将套接字传给线程对象的构造函数里。最后，对 `start` 的调用将新的线程启动。如果运行这个服务器端程序，可以一次启动若干个 Telnet 会话，每个会话都能接收数据。

清单 2.13 使用线程允许服务器一次处理多个客户端的请求

```
import java.net.*;
import java.io.*;

public class MTServer extends Thread {
    Socket csocket;
    MTServer(Socket csocket) { this.csocket = csocket; }
    public static void main(String args[]) throws Exception {
        ServerSocket ssock=new ServerSocket(1234);
        System.out.println("Listening");
        while (true) {
            Socket sock=ssock.accept();
            System.out.println("Connected");
            new MTServer(sock).start();
        }
    }

    public void run() {
        try {
            PrintStream pstream=new PrintStream(
                csocket.getOutputStream());
            for (int i=100;i>=0;i--) {
                pstream.println(i + " bottles of beer on the wall");
            }
            pstream.close();
            csocket.close();
        }
    }
}
```

```
    }  
    catch (IOException e) {  
        System.out.println(e);  
    }  
}  
}
```

向线程传递参数比较难于处理。线程对象知道调用 `start` 函数才开始执行。那意味着无需使用对象构造函数，尽管构造函数传递参数比较好。可以这样写程序：

```
MTServer svr = new MTServer();  
svr.csocket = sock;  
svr.start();
```

写成下面的形式是不可取的：

```
MTServer svr = new MTServer();  
svr.start();  
svr.csocket = sock;
```

问题在于到了 `main` 程序设置 `csocket` 时，`run` 方法可能已经在执行。除非还有别的方法来同步主线程和新线程，否则程序会崩溃。

`Thread` 类中还有其他一些方法，如下所示：

- **Sleep**——可以让线程暂停若干毫秒的时间，当该线程在睡眠时允许其他线程执行将是非常有效的。
- **Yield**——大多数系统给线程一定量的执行时间。如果线程不需要它的时间分片，可以使用 `yield` 来放弃这个分片。这在没有使用抢占式多任务系统中尤其重要。在这些系统中，只有某些调用可以切换到另一个线程。
- **SetDaemon**——可以使用 `setDaemon` 来标志一个线程是后台控制线程（`true`）还是用户线程（`false`）。只要至少有一个用户线程还在运行，Java 程序就一直运行。后台控制线程退出时并不终止程序。
- **SetPriority**——线程在系统的处理器上轮流运行。高优先级的线程要比低优先级的线程执行的多一些。在 Java 中，优先级值越大，优先级越高（这与很多操作系统中小的数代表高优先级正好相反）。优先级值的范围从 1 到 10。
- **Join**——很多情况下，一个线程需要等待另一个线程一起才能完成某个任务。这就是 `join` 方法的使用目的。当调用线程的 `join` 方法时，将一直阻塞到线程完成任务为止。可以指定一个可选的等待时间，这样就不用担心线程会永远地等待下去。

另外，在 Java 中还有支持线程的几个函数。在 `Object` 类中（所有类的基类），会发现 `wait` 和 `notify` 方法。当等待一个对象的时候，线程会一直等待，直到时间到或者线程调用了 `notify` 函数。

与线程相关的另一个话题是同步（`synchronized`）关键字，允许创建一个方法，在同一时间只允许一个线程执行该方法。假设两个线程打算调用一个带 `synchronized` 的方法。当第一个线程调用它的时候，处理方法和通常一样。然而，因为该方法是同步的，第一个线程获得

一个同步锁。当第二个线程试图调用的时候会阻塞，因为第一个线程拥有这个锁。当第一个线程执行完这个方法时，那个锁被释放出来，这样第二个线程就可以继续执行。

像前面曾提到过的一样，不一定要扩展 Thread 来创建一个线程。可以在自己的对象中实现 Runnable 接口。这样主程序会创建一个 Thread 对象，并将你的对象传给线程的构造函数。

```
Thread t = new Thread(runnableObject);
```

清单 2.14 演示了使用这个技术的完整的服务器应用。

清单 2.14 没有使用扩展 Thread 的多线程服务器，如下所示

```
import java.net.*;
import java.io.*;

public class MTServer1 implements Runnable {
    Socket csocket;
    MTServer1(Socket csocket) { this.csocket = csocket; }
    public static void main(String args[]) throws Exception {
        ServerSocket ssock=new ServerSocket(1234);
        System.out.println("Listening");
        while (true) {
            Socket sock=ssock.accept();
            System.out.println("Connected");
            new Thread(new MTServer1(sock)).start();
        }
    }

    public void run() {
        try {
            PrintStream pstream=new PrintStream(
                csocket.getOutputStream());
            for (int i=100;i>=0;i--) {
                pstream.println(i + " bottles of beer on the wall");
            }
            pstream.close();
            csocket.close();
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

如果为每一个客户端创建带有新标志的线程，将发现会浪费大量的时间来创建和销毁线程。更好的方法是使用线程池，线程只创建一次，永远不用销毁。相反，当需要的时候还可以重用这些线程。

可以将一组线程对象放在队列、栈或者其他类型的表结构里准备就绪，清单 2.26 中的程序使用链表来存放线程组，处理到来的请求。

2.2 快速解决方案

2.2.1 解析主机名

如果有一个主机名，想将它转换成 IP 地址，可以使用 `InetAddress` 类。很多情况下，同样是这个类可以将 IP 地址转换成主机名，尽管这个并不总是能用。`InetAddress` 对象不能直接构造。但是，你可以调用它的三个静态函数之一：`getLocalHost`，`getByName` 和 `getAllByName` 来构造。

一旦拥有 `InetAddress` 对象，就可以调用 `getHostName` 或者 `getHostAddress` 方法来找到相应的主机名或者 IP 地址。也可以调用 `getAddress` 方法得到一组字节形式的 IP 地址。清单 2.15 显示了使用 `InetAddress` 的一段代码。

清单 2.15 使用 `InetAddress` 查找主机名或者 IP 地址

```
try {
    address=InetAddress.getByName("www.coriolis.com");
}
catch (UnknownHostException e) {
    System.out.println("I can't find host");
    System.exit(1);
}
System.out.println(address.getHostName() + " = "
    + address.getHostAddress());
```

2.2.2 向服务器端打开 TCP 套接字

当创建套接字对象的时候，可以在构造函数里指定目标地址和端口号。可以将目标地址设为串或者 `InetAddress` 对象的形式。如果计算机有多个网络接口，也可以指定要使用的本地端口和对外接口，可以是串，也可以是 `InetAddress` 对象。

构造套接字的时候，如果主机名未知或者有其他错误发生，将会触发异常。清单 2.16 显示了打开 Web 服务器 `www.coriolis.com` 的例子。

清单 2.16 打开套接字需要异常处理代码

```
try {
    Socket sock = new Socket("www.coriolis.com",80);
}
catch (UnknownHostException e) {
    // not found
}
catch (java.io.IOException ioe) {
    // other error
```

```
}
```

2.2.3 打开服务器端套接字

要启动服务器，可以将一个 `ServerSocket` 对象实例化。它的最小的构造函数需要端口号，用于套接字监听到来的请求。同时，如果有多个网络接口时，可以为 `InetAddress` 对象指定一个具体的地址。

一旦 `ServerSocket` 实例化了，就可以使用 `accept` 方法等待连接。它将返回一个 `Socket` 对象，使得服务器端可以与客户端通信。客户端的连接只有在服务器端调用 `accept` 方法才有可能；这样，单个客户端也会连接。如果没有客户端正等待连接，此调用将会阻塞（还要受到由 `soSetTimeout` 设置的时间的限制）。

功能强大的服务器应用通常创建独立的线程来处理到来的请求，这样其他客户端就不用等待连接了。在清单 2.17 中可以看到这样的例子。

清单 2.17 `ServerSocket` 类使得接收客户端连接更容易

```
try {  
    ServerSocket ssock = new ServerSocket(2222); // port 2222  
    while (true) {  
        Socket sock=ssock.accept();  
        HandleClient(sock);  
    }  
} catch (java.io.IOException e) {  
}
```

2.2.4 创建 UDP 套接字

从某些协议或者从效率至上的角度来看，你也许愿意使用 UDP 套接字来替换 TCP 套接字。UDP 套接字并不能确保数据的传送。同时也不能保证数据的传送顺序。如果使用这种套接字，必须要有方法来处理传送数据丢失和传送顺序的问题。

通过 UDP 发送或者接收数据的时候，要使用 `DatagramPacket` 对象。这个对象允许设置 IP 地址（可以是广播地址）和端口号。它还有一个字节数组用来容纳发送或者接收的数据。还要将包对象传给 `DatagramSocket` 对象的 `send` 或者 `receive` 方法。

如果在一个服务器上处理多个事务，可以考虑调用 `DatagramSocket.connect` 来设置与远程计算机的连接。这减少了发送每个数据包给远程计算机的相关开销。但是接下来你直到调用 `disconnect` 后才能发送数据到其他计算机。

UDP 套接字不使用流。只简单地发送接收字节数组。清单 2.18 中有这样的例子。

清单 2.18 UDP 套接字使用 `DatagramPacket` 对象发送接收数据

```
byte [] ary= new byte[128];
```

```
DatagramPacket pack=new DatagramPacket(ary,128);
if (reading) {
// read
    DatagramSocket sock=new DatagramSocket(portnum);
    sock.receive(pack);
    String word=new String(pack.getData());
    System.out.println("From: " + pack.getAddress()
        + " Port: " + pack.getPort());
    System.out.println(word);
    sock.close();
} else { // write
    DatagramSocket sock=new DatagramSocket();
    pack.setAddress(InetAddress.getByName(hostname));
    pack.setData(dataString.getBytes());
    pack.setPort(portnum);
    sock.send(pack);
    sock.close();
}
```

2.2.5 向 TCP 套接字发送数据

一旦有了 Socket (套接字) 对象 (可以构造它或者从 ServerSocket.accept 调用中得到), 就可以通过调用 `getOutputStream` 来获得流, 并往流里写数据。它是包 `java.io` 的一部分, 允许往套接字里写字节 (或者是字节数组), 从而让另一方计算机读到它。

通常不会直接使用流, 而是添加一个或多个流过滤器, 用起来简单一些。例如, 可能要使用 `BufferedOutputStream` 类来缓冲字节数据, 从而提高性能。`DataOutputStream` 对象允许你写基本类型数据, `ObjectOutputStream` 对象允许写所有对象, 只要对象是可串行化的。清单 2.19 显示了一个简单的例子, 它打开一个远程计算机的套接字, 并使用 `DataOutputStream` 过滤器来发送整数数据。

清单 2.19 一个使用流发送数据的 TCP 套接字

```
void sendToSocket(String host, int port) throws Exception {
    Socket sock = new Socket(host,port);
    DataOutputStream os = new DataOutputStream(sock.getOutputStream());
    os.writeChar('X');
    os.writeChar('Y');
    os.close();
    os.sock();
}
```

2.2.6 从 TCP 套接字接收数据

要接收数据需要调用 `getInputStream` 方法。一旦拥有这个输入流对象, 就能用它读套接字中的原始字节。如果调用了 `read` 方法, 但是没有数据可读, `read` 调用会阻塞, 除非设定了

等待时间。

你可能希望使用过滤器提供额外的功能，如缓冲（`BufferedInputStream`）或者数据接收（`DataInputStream`）。清单 2.20 显示了从 TCP 套接字中接收数据的简单功能。

清单 2.20 TCP 套接字通过流来传送数据

```
void sockRcv(Socket sock) throws Exception {
    char c1,c2;
    DataInputStream is = new DataInputStream(sock.getInputStream());
    c1=is.readChar();
    c2=is.readChar();
    is.close();
}
```

2.2.7 压缩套接字数据

因为可以向输入输出流中添加过滤器，你可以相对简单地执行复杂的处理，如压缩和解压缩。`java.util.zip` 包有几个有关流的类可以用于压缩和解压缩空闲数据。

清单 2.21 中的程序演示了从套接字中接收数据，并使用 `GZIPInputStream` 来解压缩。这个服务器接收一个连接，并在事务处理完时终止连接。在命令行中使用端口号来启动这个服务器端程序。

清单 2.21 使用 Java 的压缩类来减小通过套接字发送的数据量

```
import java.net.*;
import java.io.*;
import java.util.zip.*;

public class CompRcv {
    public static void main(String[] args) throws Exception {
        ServerSocket ssock = new ServerSocket(
            Integer.parseInt(args[0]));
        System.out.println("Listening");
        Socket sock=ssock.accept();
        GZIPInputStream zip = new GZIPInputStream(
            sock.getInputStream());
        while (true) {
            int c;
            c = zip.read();
            if (c==-1) break;
            System.out.print((char)c);
        }
    }
}
```

清单 2.22 包含一个客户端程序，发送一个文件到清单 2.21 中的服务器。在命令行中使用主机名、端口号和待发送的文件来启动客户端程序。客户端和服务器端与在没有压缩数据流的

情况下工作得一样好。唯一的不同在于它们采用特殊的压缩数据的类来过滤它们的流数据。

清单 2.22 这个客户端发送压缩文件到 2.21 中的服务器

```
import java.net.*;
import java.io.*;
import java.util.zip.*;

public class CompSend {
    public static void main(String[] args) throws Exception {
        Socket sock=new Socket(args[0],Integer.parseInt(args[1]));
        GZIPOutputStream zip = new GZIPOutputStream(
            sock.getOutputStream());
        String line;
        BufferedReader bis = new BufferedReader(
            new FileReader(args[2]));
        while (true) {
            try {
                line=bis.readLine();
                if (line==null) break;
                line=line+"\n";
                zip.write(line.getBytes(),0,line.length());
            }
            catch (Exception e) { break; }
        }
        zip.finish();
        zip.close();
        sock.close();
    }
}
```

2.2.8 设定套接字的最长读时间

当从套接字中读数据时，调用通常会阻塞，直到读操作结束。如果不希望这样，可以调用 `Socket.setSoTimeout` 方法。时间的单位是毫秒，缺省值为 0（表明没有时间限制）。如果一个等待时间值，同时读操作未在指定时间内完成，读操作将抛出一个 `InterruptedException` 异常。

下面是一个例子，假设做一个调用，设定等待时间为 1 秒（1000 毫秒）：

```
Sock.setSoTimeout(1000); // may throw SocketException
```

接着，假设有一个输入流名为 `is`，可以这样写：

```
try {
    c=is.read();
}
catch (InterruptedException e) { /* Time out! */ }
```

如果发现自己使用了 `setSoTimeout`，应该考虑使用线程取代它。通常，在一个线程中处理读数据，在另一个线程中处理数据，会产生更好的解决方法。也可以使用 `getSoTimeout` 方

法将等待时间值读出来。

2.2.9 设定服务器端最长接收时间

当调用 `ServerSocket.accept` 时，程序会一直阻塞，直到有客户端连接。如果要设置等待时间（单位是毫秒），可以调用 `ServerSocket.setSoTimeout` 方法。然而，如果这样做了，应该重新考虑是否应该使用线程来设计。使用线程，可以在等待客户端连接的过程中不用停止自己的处理。可以调用 `getSoTimeout` 来读等待时间值。下面是一个例子：

```
ssock.setSoTimeout(1000); // may throw SocketException
```

这样，在 1 秒的时间内如果没有获得连接，对 `accept` 的调用将抛出一个 `IOException` 异常。

2.2.10 设定 `SoLinger`

有时候，在所有要发送的数据传送到另一台计算机完成之前可能要关闭一个套接字。这种情况下，可以调用 `Socket` 对象的 `setSoLinger` 方法来设置时间（秒）——关闭套接字之前等待数据清除的时间。可以调用 `getSoLinger` 方法来获取这个时间值。下面将这个时间值设成 5 秒，它将抛出一个 `SocketException` 异常：

```
sock.setSoLinger(true,5);
```

如果将第一个参数值设为 `false`，将使 `linger` 时间特征失效，缺省情况就是这样。很明显，这个调用只适用于 `TCP` 套接字，因为 `UDP` 套接字没有要关闭的连接。

2.2.11 设定套接字的延时行为

为使数据传输更有效，底层操作系统通常缓存数据，直到另一台计算机确认了前边的传输才发送数据包。对一些应用来说，强调交互性就不一定合适。如果想要改变这种情况，可以调用 `socket.setTcpNoDelay(true)`。可以调用 `getTcpNoDelay` 来获取当前的标志值。

提示：RFC896 定义了 Nagle 算法，该算法用来确定何时发送数据包。

2.2.12 设定保持活动选项

考虑客户端和服务端连接后，不时交换数据的情况。服务器有可能崩溃，而客户端要到下一次客户端发送数据才能觉察到。通过调用 `Socket.setKeepAlive` 方法，传递 `true` 作为参数值，套接字将定期地发送无意义的数据，只为了确认连接还是否存在。下面是例子：

```
sock.setKeepAlive(true); // may throw SocketException
```

程序看不到那些数据，但如果套接字停止通信，程序会收到一个异常，即使此时并没有传输数据。可以调用 `getKeepAlive` 方法获取当前的设置值。

2.2.13 设定缓冲区的大小

`TCP` 套接字可以缓存数据，尽管确切的细节依赖于平台。可以通过调用 `Socket.set-`

`ReceiveBufferSize` 和 `Socket.setSendBufferSize` 方法为操作系统提供建议的缓冲大小。这些值只是建议值。也可以调用 `getReceiveBufferSize` 和 `getSendBufferSize` 方法来读取缓冲的大小。下面是一个例子：

```
sock.setSendBufferSize(sock.getSendBufferSize+1024);
```

当然，这些调用可能抛出 `SocketException` 异常。

2.2.14 处理套接字异常

很多套接字操作会抛出异常。有四个常见的异常需要处理：

- `java.io.IOException`——发生在有常见 I/O 错误时。
- `java.net.BindException`——发生在请求端口正在使用时。
- `java.net.ConnectException`——发生在客户端不能连到服务器端时。
- `java.NoRouteToHostException`——发生在网络出现问题使程序找不到主机时。

通过处理 `SocketException` 异常可以捕获最后的三个异常，`SocketException` 异常是这三个异常类的基类。

2.2.15 创建多线程服务器程序

一个简单的服务器端程序包括如下基本的过程：

1. 创建一个 `ServerSocket` 类。
2. 调用 `accept`。
3. 执行必需的处理。
4. 返回并在此调用 `accept`。

然而，如果存在重要的处理，效率就不高了。最好是让服务器端使用单独的线程来处理每个客户端。最简单的方法是让自己的类来扩展 `Thread` 类。这样就可以在 `run` 方法中写自己的处理部分。在 `main` 函数（通常是静态的）里调用 `accept`，创建自己类的一个新的实例（可能要将 `accept` 返回的套接字传给它），接着调用 `start` 来运行线程。在清单 2.23 中可以找到这样的例子。

清单 2.23 一个基本的多线程服务器应用

```
import java.net.*;
import java.io.*;

public class AMTServer extends Thread {
    Socket csocket;
    AMTServer(Socket csocket) { this.csocket = csocket; }
    public static void main(String args[]) throws Exception {
        ServerSocket ssock=new ServerSocket(1234);
        while (true) {
            new AMTServer(ssock.accept()).start();
        }
    }
}
```

```
    }  
}  
  
    public void run() {  
    // client processing code here  
    }  
}
```

2.2.16 自动处理多线程服务器

因为多线程服务器应用的逻辑是可以预测的，你可能想为所有多线程服务器写一个通用的基类，那是个好的想法，但还是有问题。你想创建单个基类，使它能用常规的客户端处理来实例化你的类。然而，如果没有专门的技巧那是行不通的。

在清单 2.24 中将看到使用特殊的技术，让一个静态成员创建类的一个实例——即使那个类是 `MTServerBase` 类的一个子类。这里的方法是将一个 `Class` 对象传给 `startServer`。这个方法接着使用 `newInstance` 方法来创建对象。如果想要修改基类代码，就要假设你的子类有一个缺省的构造函数（尽管你也可以使用 `java.lang.reflect.Constructor` 中的反射函数来调用一个非缺省的构造函数）。

这就允许扩展 `MTServerBase` 类而不用提供 `startServer` 的一个常规版本。可以通过在普通名字后添加“.class”，发现有一个特定类型的类对象。如果有一个对象的实例，就可以使用 `getClass` 做相同的事情。因为向 `startServer` 方法传递类对象，能生成一个使用 `newInstance` 方法来创建对象的实例。所以，没有必要在子类中替换 `startServer` 方法。

清单 2.24 可以使用基类简单地创建多线程服务器应用

```
import java.net.*;  
import java.io.*;  
  
public class MTServerBase extends Thread {  
    // client  
    protected Socket socket;  
  
    // Here is the thread that does the work  
    // Presumably you'll override this in the subclass  
    public void run() {  
        try {  
            String s = "I'm a test server. Goodbye";  
            socket.getOutputStream().write(s.getBytes());  
            socket.close();  
        }  
        catch (Exception e) {  
            System.out.println(e);  
        }  
    }  
}
```

```

static public void startServer(int port,Class clobj) {
    ServerSocket ssock;
    Socket sock;
    try {
        ssock=new ServerSocket(port);
        while (true) {
            Socket esock=null;
            try {
                esock=ssock.accept();
// create new MTServerBase or subclass
                MTServerBase t=(MTServerBase)clobj.newInstance();
                t.socket=esock;
                t.start();
            } catch (Exception e)
                try { esock.close(); } catch (Exception ec) {}
            }
        } catch (IOException e) {
    }
    // if we return something is wrong!
}

// Very simple test main
static public void main(String args[]) {
    System.out.println("Starting server on port 808");
    MTServerBase.startServer(808,MTServerBase.class);
}
}

```

清单 2.25 显示了一个如何使用服务器基类组成另一个服务器的例子。这个专门的服务器接收命令行上输入的端口号，将小写的输入（这里是从 Telnet 程序中得到）转换成大写。可以通过输入 Control+C 或者 Control+D 来终止这个服务器应用。

注意子类中的代码只有 run 方法和一个新的 main 函数适合启动服务器。可以创建任意数目从 MTServerBase 继承而来的服务器，将问题集中在客户端的交互上。

清单 2.25 服务器使用 MTServerBase 对象

```

import java.net.*;
import java.io.*;

public class UCServer extends MTServerBase {

    public void run() {
        try {
            InputStream is=socket.getInputStream();
            OutputStream os=socket.getOutputStream();

```

```
        while (true) {
            char c=(char)is.read();
// end on Control+C or Control+D
            if (c=='\003' || c=='\004') break;
            os.write(Character.toUpperCase(c));
        }
        socket.close();
    }
    catch (Exception e) {
        System.out.println(e);
    }
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    System.out.println("Starting server on port " + n);
    startServer(n,UCServer.class);
}
}
```

2.2.17 使用线程池为客户端程序服务

对中等规模的服务端应用而言，为每个客户端创建一个线程是个好方法。但是，对较大规模的服务器端应用，经常创建和销毁线程的开销是禁止发生的。

Java 有很多种数据结构可以用来容纳线程对象。创建一个系统有好几种方法，但最简单的是构造一定数量的线程并在需要的时候仅仅使用它们去满足客户端的要求。清单 2.26 中的代码创建了一个链表，从静态的 head 变量开始。每个对象有一个 next 域指向表中的下一个线程。

可以使用 init 方法设定线程池中线程的数目。代码中很快创建了指定数目的线程。每个线程开始执行 run 方法，迅速进入 wait 状态，等待下一个 Thread 对象。

代码的剩余部分与 MTServerBase 很类似。最大的不同是当客户端连接是通过 accept 到达的，程序简单地从链表中选择第一个线程，将它从链表中删除，接着调用 notify 方法。内置的 run 方法调用 doClientProcessing 方法，这个方法会被重载用来执行有用的工作。这个方法返回时，线程将自己又放到链表上等待更多的任务。

清单 2.26 无需为每个客户端创建线程，管理一个线程池是可能的，如下所示

```
import java.net.*;
import java.io.*;

public class PoolServerBase extends Thread {
    // client
    protected Socket socket;
    // linked list of threads
```



```
static PoolServerBase head=null;
protected PoolServerBase next=null;

// start server with number of threads
static protected boolean init(Class clobj,int threads) {
    try {
        for (int i=0; i<threads; i++) {
            PoolServerBase thread = (PoolServerBase)clobj.newInstance();
            thread.next=head;
            head=thread;
            thread.start();
        }
    }
    catch (Exception e) { return false; }
    return true;
}

// add a thread to the list
static synchronized protected void addToList(PoolServerBase me) {
    me.next=head;
    head=me;
}

// hold thread until client arrives
synchronized protected void waitForSignal() {
    try {
        wait();
    }
    catch (InterruptedException e) { }
}

// main routine that schedules threads
public void run() {
    while (true) {
        waitForSignal();
        doClientProcessing();
        addToList(this);
    }
}

// Start a thread for a client
synchronized protected void handleClient(Socket s) {
    socket=s;
    notify();
}

// Here is the code that does the work
// Presumably you'll override this in the subclass
```

```
protected void doClientProcessing() {
    try {
        String s = "I'm a test server. Goodbye";
        s+="Thread: " + this.toString();
        socket.getOutputStream().write(s.getBytes());
        sleep(10000); // simulate processing
        s="Complete";
        socket.getOutputStream().write(s.getBytes());
        socket.close();
    }
    catch (Exception e) {
        System.out.println(e);
    }
}

// Test for an empty list
static synchronized protected boolean listEmpty() {
    return head==null;
}

// assign a thread to a client
static protected void assignThread(Socket sock) {
    PoolServerBase t=head;
    head=head.next;
    t.socket=sock;
    synchronized (t)
        t.notify();
}

// Start up the server
static public void startServer(int port) {
    ServerSocket ssock;
    Socket sock;
    try {
        ssock=new ServerSocket(port);
        while (true) {
            Socket esock=null;
            try {
                esock=ssock.accept();
                // wait for non-empty thread list
                while (listEmpty()) yield();
                assignThread(esock);
            } catch (Exception e)
                try { esock.close(); } catch (Exception ec) {}
        }
    } catch (IOException e) {
```

```

    }
    // if we return something is wrong!
}

// Very simple test main
static public void main(String args[]) {
    init(PoolServerBase.class,3);
    System.out.println("Starting server on port 808");
    startServer(808);
}
}

```

为了让程序便于调试，在缺省的处理中设置了 10 秒的延迟。因为每个连接会使用它的 Thread 对象，可以简单地创建若干连接，看看每个连接由不同的线程来处理。如果将线程的数目设得比较小，可以看到客户端将等待可用的线程，接着可能重用以前曾用过的线程。

清单 2.27 中列出了使用 PoolServerBase 类的例子。这个服务器在本章的前面使用多线程的例子中出现过。Telnet 到服务器，将回显字符的大写形式。输入 Control+C 或者 Control+D 会结束这个会话。

清单 2.27 这个例子使用 PoolServerBase 来管理一个线程池

```

import java.net.*;
import java.io.*;

public class UCServerPool extends PoolServerBase {

    protected void doClientProcessing() {
        try {
            InputStream is=socket.getInputStream();
            OutputStream os=socket.getOutputStream();
            while (true) {
                char c=(char)is.read();
                // end on Control+C or Control+D
                if (c=='\003' || c=='\004') break;
                os.write(Character.toUpperCase(c));
            }
            socket.close();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        init(UCServerPool.class,10);
        System.out.println("Starting server on port " + n);
        startServer(n);
    }
}

```

第3章 简单协议

3.1 深入介绍

我当了接近 25 年的无线电操作员。另外，我还教了很多来自世界各地的学生。因此我也痛苦地了解到美国公民的声誉。你该知道我在说些什么。如美国人不会为学习公制系统而烦恼（因为他们使用的就是公制系统）。美国人也不喜欢处理别的民族的货币。

语言是一个特别让人困窘的问题。极少数的美国人（包括我自己）能说两种语言。更糟糕的是，绝大多数美国人认为如果你大声尖叫，还以为你在向周围的人说另外一种语言呢。

当然，美国很强大。很多母语不是英语的人花时间来学英语也就不足为奇了。当然，这只让问题变得更糟。毕竟，如果经常发现外国人说自己的语言，为什么还要学习他们当地的方言呢？

通过网络与计算机通信与之类似。通信的两端需要相同的语言。流行的协议，像超文本传输协议（HTTP）等要求很多客户端都有它们自己的语言。不常用的服务也会强制使用它们自己特殊的语言（就像使用方言一样）。

在上一章里，学习了如何打开一个套接字并创建客户端可以使用的套接字。但是，这只是战斗胜利的一半。你还得掌握与另一端通信的语言。

在处理更多有用的协议之前，看看常用的一些简单的协议是一个好方法。这些协议不会像 HTTP 那样有吸引力，但是如何使用这些协议将会应用于以后编写更加复杂的服务器端程序。

特别地，本章有如下内容：

- Echo 服务器端和客户端主要用于网络上的测试
- 设置整个网络上的时间协议
- 用于了解用户和机器的协议

有一个在 Java 中看不到的独特的协议是 ping 协议。那是因为 ping 使用了网际控制消息协议，这个协议 Java 并不直接支持它。

3.1.1 Echo 协议

Echo 协议（由 RFC862 定义）使用了 TCP 协议或者 UDP 协议（端口 7），是最容易理解的协议之一。上一章中一些服务实际上就是 echo 服务。一个 echo 服务端只是简单地返回发送给它的数据。

关于 echo 一件有趣的事是 echo 服务器端对客户端并不是很敏感。例如，假设客户端发送三行测试消息到 echo 服务器端。如果客户端用换行符终止消息，echo 服务器也会这样终止。如果客户端使用回车并换行，那么 Echo 服务器也返回那些值。实际上服务器做这件事并没有任何逻辑。通常说 echo 服务器能用多种协议，就像说录音机知道所有语言一样。

一个 TCP echo 服务器是微不足道的，因为它的连接是由 TCP 套接字来创建的。当客户端连向它的时候，服务器简单地将接收过来的每一样数据从相同的套接字上发送回去。但是 UDP 就要难处理一些。你不得不将数据从 UDP 数据报中提取出来，找到源 IP 地址和端口号。接着要组成另一个数据报，回复给源发者。

如果服务器在运行一个 echo 服务，可以使用 Telnet 客户端来连接并测试它。当然，Java 程序可以简单地发送和接收应答。在清单 3.1 里可以找到一个 TCP echo 服务端应用。

清单 3.1 本 echo 服务端应用使用了多线程和 TCP 套接字

```
import java.io.*;

public class TcpEchoServer extends MTServerBase {
    public void run() {
        try {
            InputStream is = socket.getInputStream();
            OutputStream os = socket.getOutputStream();
            int c=0;
            while (c!=-1) {
                c=is.read();
                if (c!=-1) os.write(c);
            }
        } catch (IOException e) {
            System.out.println(e);
        }
        try {
            socket.close();
        } catch (IOException e) {}
    }

    public static void main(String [] args) {
        startServer(7,TcpEchoServer.class);
    }
}
```

UDP 服务端程序不像你想象的那样类似。因为每个 UDP 包是独一无二的，没有必要创建多线程。只需要简单地处理到来的包就可以。DatagramPacket 对象包含发送请求的程序的地址和端口号。因此可以准确地使用相同的包来返回数据给发送方。在清单 3.2 中可以找到相关代码。

清单 3.2 一个也能使用 UDP 套接字的 echo 服务器端程序

```
import java.net.*;
import java.io.*;

public class UdpEchoServer {
    static final int BUFFERSIZE = 256;
    public static void main(String [] args) {
        DatagramSocket sock;
        DatagramPacket pack=new DatagramPacket(
            new byte[BUFFERSIZE],BUFFERSIZE);
        try {
            sock=new DatagramSocket(7);
        }
        catch (SocketException e) {
            System.out.println(e);
            return;
        }
        // echo back everything
        while (true) {
            try {
                sock.receive(pack);
                sock.send(pack);
            }
            catch (IOException ioe)
                System.out.println(ioe);
        }
    }
}
```

还需要一个专门的客户端程序来测试 UDP 服务器端程序。你可以在清单 3.3 中找到相关代码。因为没有连接，客户端能一直等待直到有响应。为防止这种情况发生，代码调用了 `setSoTimeout` 函数，这样最长等待时间在 5 秒钟（5000 毫秒）。如果在这段时间之内还没有响应，`read` 函数就抛出一个 `InterruptedIOException` 异常。

echo 服务端的一个可能的用途是用来估计网络连接的速度。清单 3.3 中的客户端程序估计了往返时间。当然，这个程序中使用了简化的方法，它忽略了创建 `Calendar` 对象所产生的影响，但它给出了在不同主机之间衡量相对速度的一个方法。

清单 3.3 这个客户端程序估计了到服务器端的网络连接速度

```
import java.net.*;
import java.io.*;
import java.util.Calendar;
public class UdpEchoClient {
    static final String testString = "Greeks bearing gifts";
    public static void main(String [] args) {
```

```

InetAddress address;
try {
    address = InetAddress.getByName(args[0]);
}
catch (UnknownHostException host) {
    System.out.println(host);
    return;
}
DatagramPacket pack=new DatagramPacket(
    testString.getBytes(),testString.length(),
    address,7);
DatagramPacket incoming=new DatagramPacket(
    new byte[256],256);
DatagramSocket sock=null;
try {
    Calendar start, end;
    sock = new DatagramSocket();
    start = Calendar.getInstance();
    sock.send(pack);
    sock.setSoTimeout(5000);
    sock.receive(incoming);
    end = Calendar.getInstance();
    String reply = new String(incoming.getData());
    reply=reply.substring(0,testString.length());
    if (reply.equals(testString)) {
        System.out.println("Success");
        System.out.println("Time = " +
            (end.getTime().getTime()-start.getTime().getTime()) +
            "mS");
    }
    else
        System.out.println("Reply data did not match");
}
catch (SocketException socke) {
    System.out.println(socke);
}
catch (IOException ioe) {
    System.out.println(ioe);
}
finally {
    sock.close();
}
}
}

```

这个程序有个奇怪的地方在于它不能区分具体的请求和广播的请求。因此，如果运行服务器端程序，接着使用广播地址（如 255.255.255.255）调用客户端程序，它将对任意其他机器上的复制内容进行响应。

你可以运行两个不同的 echo 服务器端程序。一个用于处理 TCP，另一个处理 UDP。然

而，真实的 echo 服务器端程序应该同时监听 TCP 和 UDP 请求。可以写一个类将两个服务器端程序代码合并成一个单一的类，从而简单地将两个服务合并在一起（参考清单 3.4）。

清单 3.4 这个类将 TCP 和 UDP 服务器端应用合并到一个程序里

```
class EchoServerUdp extends Thread {
    public void run() {
        UdpEchoServer.main(null);
    }
};

public class EchoServer extends TcpEchoServer {
    public static void main(String [] args) {
        new EchoServerUdp().start(); // start UDP thread
        TcpEchoServer.main(args);
    }
}
```

类 EchoServerUdp 没有定义成公有类，因此它只能在 EchoServer 类文件中起作用。它仅代表一个启动对象 UdpEchoServer 的线程。而 EchoServer 对象本身只是扩展了类 TcpEchoServer。它唯一做的就是替换了 main 函数。新的 main 函数启动 UDP 线程，接着将控制返回到原始的 main 函数。它允许在相同的端口打开 UDP 和 TCP 套接字。

一个更简单的协议“the discard service”在 RFC863 里有介绍。这个服务使用了端口 9 (TCP 或者 UDP)，将发给它的任何数据全部丢弃。

3.1.2 Finger

Finger 协议（在 RFC1288 中定义，在 RFC1196 和 RFC742 中更新）对 Unix 用户来说应该非常熟悉，但可能对其他操作系统中拥有超级用户的人不是很熟悉。基本思想是客户端可以探测（finger）一台主机以找到有关那台主机的信息（典型的是当前记录在机器里的用户列表）。另外，你能探测到特别的用户并获得该用户的信息。一台典型的 Unix 主机将显示用户基本信息（例如用户的真名和位置）和用户计划文件中的信息。其他操作系统可能显示其他信息（或者甚至没有信息）。不同的主机可能选择性地提供其他信息作为对 finger 响应。Finger 协议实际上并不保证 finger 返回什么样的数据。

Finger 服务使用 TCP，端口为 79。客户端发送一行请求。服务器端响应将关掉连接。RFC 中规定行尾还有回车符和换行符。同时它推荐客户端过滤掉非 ASCII 数据。

为了查找完整的一个主机，客户端只发送空行。如果想要查找特殊的用户，那一行应该包含用户的 ID。也允许通过将多个用户 ID 放到同一行中并以空格分开，来查询多个用户。RFC 规定可以在命令行中增加/W 选项来获得详细输出，尽管实际上如果这样做服务器会忽视这个选项。

除了向一台机器查询用户，还可以请求一台机器在主机名前加@使用 finger 向另一台机器查询。因为几个有名的蠕虫程序使用过 finger，很多站点不再运行 finger 服务，其中有很多限制了 finger 的使用。例如，服务器很少允许向其他机器提交请求。即使提交空的请求，很多公用服务器也不会将所有的用户列出。RFC 要求对不接受的请求返回“拒绝”值，也要求在处理请求的时候至少要显示用户的全称。但是实际上，客户端很少期望得到特别的数据格式。

Finger 客户端程序较容易编写（参看清单 3.5）。由于对返回的数据不太确定，客户端代码只是将返回的数据当作串来处理。调用程序（例如，main 函数）如果发现返回的数据比较合适，可以将它格式化并显示出来。

因为不同的平台使用不同的行结束组合，客户端代码明确地使用 \n 来与规定的标准集相匹配。另外，因为只有 ASCII 字符允许使用，代码使用了特定的字节编码，即通过向 getBytes 方法传递编码名（ISO8859_1）。

Encoding（编码）

getBytes 方法以及其他使用字节和字符的方法，可以依靠编码串来确定如何在 Unicode 字符和字节之间转换。这是必需的，因为 Unicode 字符需要用两个字节表示。这一章的程序使用了 ISO8859_1 编码——Java 支持的几种编码方法之一。

下面是 Java 支持的几种基本编码方法：

- ASCII——信息交换美国标准码。
- Cp1252——Windows 拉丁-1 码 (Latin-1)。
- ISO8859_1——1 号拉丁字母。
- UnicodeBig——16 位 Unicode 转换码 (UTF)，采用大端派 (big-endian) 字节顺序，带有字节标志。
- UnicodeLittle——16 位 UTF，采用小端派 (little-endian) 字节顺序。
- UnicodeLittleUnmarked——不带标识的 16 位 UTF，小端派 (little-endian) 字节顺序。
- UTF8——8 位 UTF 码。
- UTF-16——16 位 UTF，字节顺序由强制的初始字节顺序标识来确定。
- 到 HTML3.2（包括 HTML3.2 本身）的时候，HTML 规范开始要求使用 ISO 8859_1 字符集。

当然，很多浏览器并不遵守这个要求，后来的 HTML 版本还允许使用其他的字符集。因此，对于 Web 程序，使用 ISO8859_1 通常是安全的。其他的 Internet 协议没有强制用户必须支持哪些字符集。最安全的应该是 ASCII 字符集，但是很多主机也认可 ISO8859_1 字符集。

另外，Java 的国际版本可以支持很多其他编码方式，在“快速解决方案”那一节里可以找到这些编码方式的详细介绍。

清单 3.5 这个 finger 客户程序允许查询一个远程主机

```
import java.net.*;
import java.io.*;

public class Finger {
    public String finger(String host, String users)
```



```

throws IOException, SocketException {
    String outstring="";
    int c=0;
    Socket sock=new Socket(host,79);
    OutputStream os = sock.getOutputStream();
    InputStream is = sock.getInputStream();
    users = users + "\r\n";
    os.write(users.getBytes("iso8859_1"));
    try {
        while (c!=-1) {
            c=is.read();
            if (c!=-1) outstring+=(char)c;
        }
    }
    catch (IOException e) {}
    return outstring;
}

public static void main(String[] args) {
    String hostname = "";
    String ulist = "";
    if (args.length==0) {
        System.out.println("usage: finger host [user...]");
        System.exit(1);
    }
    if (args.length>=2)
        for (int i=1; i<args.length; i++)
            ulist+=args[i]+" ";
    hostname=args[0];
    Finger worker = new Finger();
    try {
        System.out.println(worker.finger(hostname,ulist.trim()));
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
}

```

这个对象在其他 Java 程序中很容易使用。例如，清单 3.6 显示了一个 JSP 程序使用该对象提供一个基于 Web 的 finger 网关（参看图 3.1）。要测试该程序，需要一个 JSP 的 Web 服务器，如 Tomcat 或者 Allaire 公司的 JRun。同时，还需要将编译好的 Finger.class 文件放到该 Web 服务器的 class（类）路径里。如果 Web 服务里有 JSP 应用程序路径，可以使用 WEB-INF/classes 目录作为 class 路径，尽管实际情况可能随着系统的设置不同以及使用的是哪个服务器而有所不同。

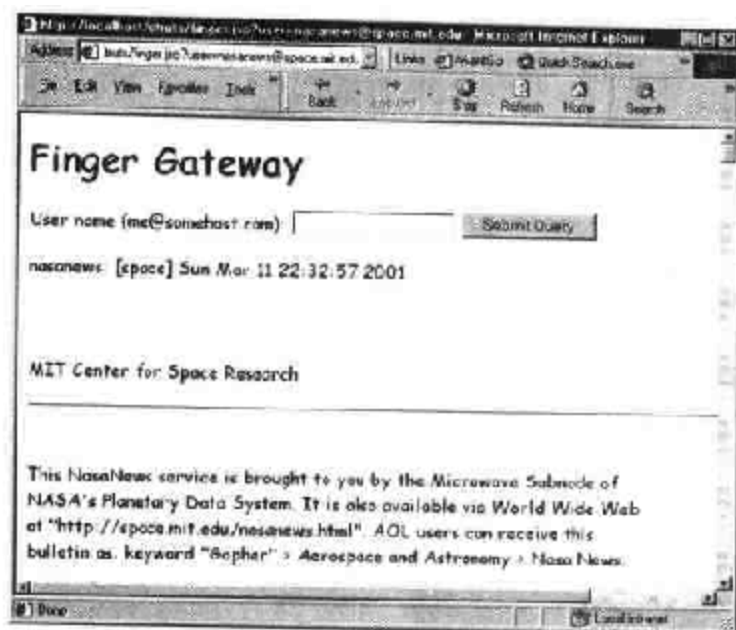


图 3.1 一个基于 Web 的 finger 网关

清单 3.6 这个 JSP 程序提供了一个基于 Web 的 finger 网关

```
<H1>Finger Gateway</H1>
<FORM SUBMIT=finger.jsp METHOD=GET>
User name (me@somehost.com): <INPUT NAME=user>
<INPUT TYPE=SUBMIT>
</FORM>
<%
if (request.getParameter("user")!=null) {
    Finger fing = new Finger();
    String inp = request.getParameter("user");
    String u;
    String host;
    int at = inp.indexOf('@');
    if (at== -1) {
        u="";
        host=inp;
    }
    else {
        host=inp.substring(at+1);
        u=inp.substring(0,at);
    }
    String s;
    try {
        s = fing.finger(host,u);
    }
    catch (Exception sexc) {
        s="Unknown host or network error";
    }
}
```

```

int n0=0;
do {
    int n=s.indexOf('\n',n0);
    if (n!=-1)
        out.print(s);
        break;
    }
    out.print(s.substring(n0,n) + "<BR>");
    n0=n+1;
} while (n0!=s.length());
}
%>

```

你将注意到 JSP 允许输入典型的 email 地址（例如alw@coriolis.com）。代码自动提取主机名和用户 ID 传递给 finger 方法。脚本使用
来作换行标志，这样在浏览器里显示起来会好一些。这个 JSP 文件是自包含的，它提交数据给相同的 JSP 页，并作出合适的处理。这对开发者很方便，因为只需处理一个文件，同时对用户也很友好，因为用户可以向同一个页面提交新的请求，同时还可以看到前一个请求的数据。

要求使用一个常规 finger 服务可能有几个原因。例如，你可能想使用 finger 来获得网络的状态。连向真实世界传感器的计算机可能使用 finger 使得网络上其他计算机能访问它的数据。服务器端程序比较难写，因为必须接受或者拒绝协议定义的不同请求。清单 3.7 显示了基于第 2 章 MTServerBase 对象的完整的服务器端实现。

清单 3.7 这个 finger 服务器使用了属性文件来控制它的响应

```

import java.io.*;
import java.util.*;
import java.net.*;

public class FingerServer extends MTServerBase {
    static Properties prop=null;
    public FingerServer() {
        if (prop==null) {
            prop = new Properties();
            try {
                prop.load(new FileInputStream("FingerServer.prop"));
            }
            catch (IOException e) {
                System.out.println("FingerServer.prop not present");
            }
        }
    }

    void socketClose() {
        try {

```

```
        socket.close();
    }
    catch (IOException e) {}
}

public void run() {
    BufferedReader is;
    OutputStreamWriter os;
    String s;
    String cmd;
    try {
        is = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        os = new OutputStreamWriter(socket.getOutputStream(),
            " iso8859_1");
    }
    catch (Exception ioe) {
        System.out.println("Can't create streams\n" + ioe);
        socketClose();
        return;
    }
    // read command line
    try {
        cmd=is.readLine();
    }
    catch (IOException e) {
        socketClose();
        return;
    }

    // look up entry
    if (cmd.equals(""))
        cmd="*default*";

    // parse multiple requests
    StringTokenizer tok = new StringTokenizer(cmd);
    String token;
    while (tok.hasMoreTokens())
        token=tok.nextToken();
        if (token.equals("/W")||token.equals("/w"))
            continue;
        if (token.indexOf('@')== -1)
            s=prop.getProperty(token,"Not Found");
        else
            s="Finger forwarding service denied";
    // write it out
    try {
        s+="\r\n";
    }
```

```

        os.write(s,0,s.length());
        os.flush();
    }
    catch (IOException we) {
        System.out.println("Write exception: " + we);
    }
}

socketClose();
}

public static void main(String [] args) {
    startServer(79,FingerServer.class);
}
}

```

FingerServer 对象提供了普通的构造函数。这个构造函数打开一个属性文件，名为 **FingerServer.prop**（见清单 3.8）。这个文件包含与用户名相对应的关键字以及对请求作出响应的返回值。服务器端拒绝提交请求。如果客户端发出空请求，服务器端返回缺省属性。

另外，服务器使用的是 ISO8859_1 编码，使用 **readLine** 函数从客户端那里接收命令，接着 **StringTokenizer** 将命令行切分用于处理。主机忽略/W 这一选项（为安全起见也检查/w）。任何包含@字符的标志将产生拒绝的消息，因为服务器不支持提交请求。

清单 3.8 这个属性文件为 **finger** 服务器定义了两个用户以及一个缺省的响应

```

alw=Al Williams, League City TX
raya=Ray Argus, Houston TX
*default=This is the example Java finger server at AWC.

```

有关服务器有趣的一点是它们必须提供信息，但是信息实际上不一定依赖于服务器。这个思想源于 **n 层（tier）** 系统和代理服务器（在第 13 章会有更多描述）。一个服务器可以将相应职责委任给其他服务器。

清单 3.9 列出了运用上述思想的 **finger** 服务器端程序。仅仅将 **finger** 请求提交到在命令行中指定的计算机中去。这就是代理服务器的使用方式。Java 服务器充当另一台机器上真正服务器的代理。要连接真服务器，程序使用了清单 3.6 中的 **Finger** 类。

清单 3.9 这个类充当一个代理，将 **finger** 请求提交给远程服务器

```

import java.net.*;
import java.io.*;

public class FingerProxy extends MTServerBase {
    static String parentHost;

    void socketClose() {
        try {

```

```
        socket.close();
    }
    catch (IOException e) {}
}

public void run() {
    Finger fing = new Finger();
    BufferedReader is;
    OutputStreamWriter os;
    String s;
    String cmd;
    try {
        is = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        os = new OutputStreamWriter(socket.getOutputStream(),
            "iso8859_1");
    }
    catch (Exception ioe) {
        System.out.println("Can't create streams\n" + ioe);
        socketClose();
        return;
    }
    // read command line
    try {
        cmd=is.readLine();
    }
    catch (IOException e) {
        socketClose();
        return;
    }

    try {
        s=fing.finger(parentHost,cmd);
        os.write(s,0,s.length());
        os.flush();
    }
    catch (IOException we) {
        System.out.println("Write exception: " + we);
    }
    socketClose();
}

public static void main(String [] args) {
    parentHost=args[0];
    startServer(79,FingerProxy.class);
}
}
```


中途截取请求可以有多个用途，如可以调试一个连接。一些充当两台机器之间的仲裁者的代理服务器，由于安全方面的限制，通常不能互相通信。Finger 代理可能不是你见到的最实用的代理，但相对其他协议而言，代理是非常重要的。

3.1.3 Whois 协议

另一种能提供有关人的信息的协议就是 whois 协议。whois 与 finger 最大的不同在于有关 whois 的信息主要位于几个集中式服务器当中。特别地，whois.internic.net 是用来识别域名数据的主服务器。然而，也有其他一些服务器提供 whois 数据。每个域名登记员都提供他们自己的 whois 服务器。例如，网络解决方案在 whois.networksolutions.com 有一个 whois 服务器。

来自 whois 服务器的数据是没有规范化的，其格式对人可读，而且类似于地址簿。原始的规范出现在 RFC954 里。随着 Internet 的快速增长和多个登记员的出现，需要更健壮的系统，有人建议用 RFC1913 和 RFC1914 里定义的系统来替换 whois，但是这个新的系统并未被广泛使用。

Whois 协议与 finger 协议没有太大的不同。只须为 whois 服务器（通常是 whois.internic.net）在 43 号端口打开 TCP 套接字。接着，发送单行（由 \n 结束）命令给服务器。这一行包括名字、名字列表、名字的一部分或者是域名。

Whois 命令的输出随服务器不同和时间的改变而不同，即使在同一服务器上（命令集也不是特别的固定）。对 Internic 服务器而言，每个实体都有一个句柄（例如我的实体句柄是 AW1127）。在这个句柄上查询将找到精确的匹配。任何其他方式都可能显示多个匹配。

你可以在查找串的前边增加一个或者多个前缀以减少输出的量（见表 3.1）。例如，可以往搜索串中加入域（domain）这个词，以限制输出，结果只输出域名。不同的服务器可能认可或者忽视不同的命令。可以使用 Telnet 登录服务器，发一个 help 命令，看看具体的服务器的命令。

表 3.1 常用的 whois 命令前缀

前缀	作用
domain	限制为主机
handle	限制为句柄
nameserver	限制为名字服务器
registrar	限制为登记员
expand	显示单个记录的完全匹配
full	长段显示每个记录
host	限制为主机
mailbox	限制为 email 地址（或者在查询里使用@）

续表

前缀	作用
organization	限制为组织
partial	匹配记录的开始部分 1
person	限制为人
server	限制为服务器
summary	显示摘要
=	同 Expand
~	从不显示子一级的内容（与 Expand 相反）
!	同 handle
\$	同 summary

清单 3.10 列出了实现 whois 客户端的一个类。在缺省情况下，main 例程使用网络解答服务器，但通过在第一个命令行参数前加入%字符来命名服务器，可重载这个例程的处理。

清单 3.10 这个 whois 客户端缺省指向 whois.internic.net 服务器

```
import java.net.*;
import java.io.*;

public class Whois {
    public void whois(String query) throws IOException {
        whois(query, "whois.internic.net");
    }

    public String whois(String query, String server) throws IOException {
        Socket sock = new Socket(server, 43);
        int c=0;
        String outstring="";
        OutputStream os = sock.getOutputStream();
        InputStream is = sock.getInputStream();
        query += "\r\n";
        os.write(query.getBytes("iso8859_1"));
        try {
            while (c!=-1) {
                c=is.read();
                if (c!=-1) outstring+=(char)c;
            }
        }
        catch (IOException e) {}
        return outstring;
    }
}
```

```
}  
public static void main(String[] args) {  
    String hostname = "";  
    String ulist = "";  
    if (args.length==0) {  
        System.out.println("usage: whois [%host] query");  
        System.exit(1);  
    }  
    int argn=0;  
    hostname="whois.networksolutions.com"; // default  
    if (args.length>1 && args[0].charAt(0)=='%') {  
        hostname=args[argn].substring(1);  
        argn++;  
    }  
    for (int i=argn; i<args.length; i++)  
        ulist+=args[i]+" ";  
  
    Whois worker = new Whois();  
    try {  
        System.out.println(worker.whois(ulist.trim(),hostname));  
    }  
    catch (Exception e) {  
        System.out.println(e);  
    }  
}
```

3.1.4 基本时间协议 (Basic Time)

拥有一个计算机网络时，如果系统的当前时间不一致，将很令人烦恼。具有错误时戳的文件可能造成程序错误，并对任何文件的同步任务造成破坏。

获取网络上的一种服务，其中一种方法是得到另一台计算机的时间。如果网络上所有机器都同步成一个时间，那些有问题的机器将不会产生那么多的问题。

实际上，RFC 没有定义时间协议的单一服务。它定义了两个不同的服务：RFC867 使用未定义格式提供了对人可读的时间协议服务，而 RFC868 以机器可读的形式提供时间协议服务。

RFC867 要求服务器在 TCP 或者 UDP 的 13 号端口上将时间发送给任何调用者。因为时间的格式没有具体的规定（RFC 建议有两种常用格式），而只不过要求一个 finger 客户端或者服务器端作适当改变，并报告时间。

有几个有名的服务器将它们的时间与一个原子时钟保持同步，并且为网络的参考时间服务（见表 3.2）。其中一些服务器是初始服务器，另外一些是二级服务器，它们与初始服务器同步。在大多数情况下，二级服务器一样好，并且比初始服务器更容易获取服务。

表 3.2 一些时间服务器

URL	IP 地址	位置
tock.usno.navy.mil	192.5.41.41	U.S. Navy
tock.usnogps.navy.mil	204.34.198.41	U.S. Navy
time-a.nist.gov	129.6.15.28	NIST, Gaithersburg, Maryland
time-b.nist.gov	129.6.15.29	NIST, Gaithersburg, Maryland
time-a.timefreq.bldrdoc.gov	132.163.4.101	NIST, Boulder, Colorado
time-b.timefreq.bldrdoc.gov	132.163.4.102	NIST, Boulder, Colorado
time-c.timefreq.bldrdoc.gov	132.163.4.103	NIST, Boulder, Colorado
utcnist.colorado.edu	128.138.140.44	University of Colorado, Boulder
time.nist.gov	192.43.244.18	NCAR, Boulder, Colorado
time-nw.nist.gov	131.107.1.10	Microsoft, Redmond, Washington
nist1.datum.com	63.149.208.50	Datum, San Jose, California
nist1.dc.glassey.com	216.200.93.8	Abovenet, Virginia
nist1.ny.glassey.com	208.184.49.9	Abovenet, New York City
nist1.sj.glassey.com	207.126.103.204	Abovenet, San Jose, California
nist1.aol-ca.truetime.com	207.200.81.113	TrueTime, AOL facility, Sunnyvale, California
nist1.aol-va.truetime.com	205.188.185.33	TrueTime, AOL facility, Virginia

一般来说，你发往主机的数据是无关紧要的（使用 Telnet 连接表 3.2 里的服务器中的 13 号端口，然后就会知道我的用意）。不幸的是，返回的数据格式差异很大。下面是来自美国海军天文台服务器的一个输出：

```
Fri Mar 9 14:28:06 2001
```

再看看，下边是一分钟以后来自 NIST（国家标准技术研究所）的结果：

```
51977 01-03-09 14:29:14 74 0 0 651.2 UTC (NIST) *
```

NIST 将它的格式发布在其网站上。下面是对其中一些元素的解释：

- 51977——它是修改了的 Julian 日。修改过意味着它是实际 Julian 日的后 5 位，通常开始于公元前 4713 年 1 月 1 日。用这个数加上 2,400,000，将产生正确的 Julian 日。
- 01-03-09——表示年，月，日格式的日期。
- 14:29:14——时，分，秒格式的时间，这个时间总是同等通用时间（UTC）。
- 74——这个代码指示在异常情况下美国是否遵循夏令时。如果代码为 00，夏令时无效，如果代码为 50，夏令时起作用。如果当前日期处于一个月的夏令时变化范围内，

这个数目将指示余下的天数。因此，在本例当中，离夏令时开始有效还有 23 天（ $50+23+1=74$ ）。

- 0——第一个 0 指在这个月末将不会有闰秒来纠正时钟。这个域值为 1，则指这个月末将有一个闰秒加到时钟里；若为 2，则要从时钟里减去一个闰秒。
- 0——第二个 0 指服务器当前的“健康”状况。0 指服务器操作恰当；如果域值为 1，则时间会有高达 5 秒钟的误差；若为 2，则指可能有超过 5 秒的误差；若为 4，则指出现了错误，时间可能整个都是错误的。
- 651.2——这是加到时间上的毫秒数，主要是减少网络延迟（由 TCP/IP 传播造成，它有较长的延迟）。
- UTC (NIST)——这个时区指示器指示了同等通用时间 (Coordinated Universal Time)。你可能想知道为何 UTC 是同等通用时间的简写，当国际电信联盟为 UTC 标准化时，美国人想使用 CUT 作为它的简写，然而法国人想使用 TUC 以反映法语中的词序，因为没有一方愿意妥协，结果 UTC 成为折衷的结果。在过去，UTC 以格林威治时间 (GMT) 或者祖鲁 (zulu) 时间而出名。
- *——当你打电话时请求时间服务时，话务员会说：“时间将是...”，这个星号是指语调——为了实用。当你接收到这个字符时，指示的时间就是合法的。当然，如果发生网络延迟和其他问题时，它将后延一位，但它将会被关闭。

清单 3.11 是这个协议的一个简单的客户端程序。比前边的 finger 客户端程序多不了多少，缺省情况下，程序使用美国海军气象台服务器，但你也可以在命令行里使用不同的服务器名来改变它，因为时间服务器无需任何输入，没有必要让客户端发送任何东西。简单的创建连接就足以使用 TCP 来获取时间数据。

也可以使用 UDP（也在 13 号端口上）执行相同的任务，但那样还需要发送无意义的数据包给服务器。这个包允许服务器应答，因为 UDP 没有连接到如下信号——客户端正在请求数据。

清单 3.11 这个服务器程序使用 RFC867 中的协议来获取时间

```
import java.net.*;
import java.io.*;

public class Time867 {
    public static void main(String[] args) {
        String hostname = "tock.usno.navy.mil";
        if (args.length==1) hostname=args[0];
        try {
            int c;
            Socket sock=new Socket(hostname,13);
            InputStream is = sock.getInputStream();
            do {
```



```

        c=is.read();
        if (c!=-1) System.out.print((char)c);
        } while (c!=-1);
    }
    catch (IOException e) { System.out.println(e); }
}
}

```

写这个服务器端程序带一点技巧，只因为 Java 提供的类 `Date` 很不好使用。类 `DateFormat`（来自包 `java.text`）允许你将一个 `Date` 对象格式化，服务器使用这个类产生所需要的串。如果你是一个坚持己见的人，应该注意到 Java 中处理 `dates` 的类通常不会考虑闰秒的问题，因此这个服务器不会像 NIST 服务器那么精确。然而，对于大多数应用来说，它已经足够精确了。

尽管这个服务器只处理 TCP 请求，我们还是能很容易将它修改成 UDP 的 `echo` 服务器，执行相同的任务（见清单 3.12）。接着你可以将两个服务器端程序连接在一起，就像清单 3.4 中的两个 `echo` 服务器程序一样。

清单 3.12 这个服务器程序使用 TCP 为 RFC867 中的客户端提供了时间数据

```

// Server for RFC867 time (TCP only)
import java.io.*;
import java.util.*;
import java.text.DateFormat;

public class Server867 extends MTServerBase {
    public void run() {
        try {
            OutputStream os = socket.getOutputStream();
            Date cal = new Date();
            DateFormat df=DateFormat.getDateTimeInstance(
                DateFormat.FULL,DateFormat.FULL);
            df.setTimeZone(TimeZone.getTimeZone("GMT"));
            String output=df.format(cal)
                +"\r\n";
            os.write(output.getBytes("iso8859_1"));
            os.flush();
            socket.close();
        }
        catch (IOException e) {}
    }
    public static void main(String[] args) {
        startServer(13,Server867.class);
    }
}

```

RFC868 定义的另一种服务对于机器来说更实用，在这个协议里，与端口 37（TCP 或者 UDP）相连的客户端将以 32 位数的格式获得当前日期和时间，这个数代表着从 1900 年 1 月 1 日午夜算起的秒数。

注意：到 2036 年，这个服务就会过期，可能会引发另一个类似于 2000 年问题的危机。当然，那时我将 73 岁了，希望已经退休！

Date 对象使用 `getTime` 可以返回自 1970 年 1 月 1 日午夜算起的毫秒数，因此这儿会有一点误差。为实用起见，可以将这个结果除以 1000（得到秒数）接着加上 2,208,988,800 秒（考虑闰年后的 70 年的秒数）。在清单 3.13 中可以发现一个服务器的例子（使用的是 TCP）。

清单 3.13 这个类实现了 RFC868 中的时间服务器

```
// Server for RFC868 time
import java.io.*;
import java.util.*;
import java.text.DateFormat;

public class Server868 extends MTServerBase {
    public void run() {
        try {
            OutputStream os = socket.getOutputStream();
            long tick = new Date().getTime()/1000;
            byte [] outarray = new byte[4];
            tick+=2208988800L;
            outarray[0]=(byte)((tick & 0xFF000000L)>>24);
            outarray[1]=(byte)((tick & 0xFF0000L)>>16);
            outarray[2]=(byte)((tick & 0xFF00L)>>8);
            outarray[3]=(byte)(tick & 0xFFL);
            os.write(outarray);
            os.flush();
            socket.close();
        }
        catch (IOException e) {}
    }
    public static void main(String[] args) {
        startServer(37,Server868.class);
    }
}
```

但对客户端而言，出现了一个独特的问题，Java 不善于处理无符号运算。当你试图将从服务器那儿接收到的字节数据转化为 long 型数据时，会碰到负数的麻烦，它将引起各种问题。

一个可能的解决方法见程序清单 3.14，程序将数据的最高位去掉，以确保结果为正值。接着，在调整 1900 到 1970 之间的时间值时，程序要确定最高位值是否设置好，如果设置好了，就从调整值那里减去 0x80000000。这样可以取得相同的效果，还能避免偶然产生负值。

清单 3.14 可以使用这个客户端程序从 RFC868 中的服务器那里取得时间值。

```
import java.util.*;
import java.net.*;
import java.io.*;
```

```

import java.text.DateFormat;

public class Client868 {
    public static void main(String [] args) {
        Socket sock=null;
        try {
            sock=new Socket(args[0],37);
            byte [] inarrayb=new byte[4];
            int [] inarray=new int[4];
            Date date=new Date();
            long tick;
            InputStream is=sock.getInputStream();
            is.read(inarrayb);
            // convert array to integers
            for (int i=0;i<4;i++) {
                inarray[i]=inarrayb[i];
                inarray[i]&=0xFF;
            }
            int tmp=inarray[0];
            // make positive
            inarray[0]&=0x7F;
            tick=(inarray[0]<<24)+(inarray[1]<<16)+(inarray[2]<<8)+inarray[3];
            long adj=2208988800L;
            // if original number was >=0x80000000, adjust the adjustment
            if ((tmp&0x80)==0x80) {
                adj-=0x7FFFFFFF;
                adj~;
            }
            tick-=adj; // convert between 1900/1970
            tick*=1000; // convert to ms
            date.setTime(tick);
            DateFormat df=DateFormat.getDateTimeInstance(
                DateFormat.FULL,DateFormat.FULL);
            System.out.println(df.format(date));
        }
        catch (IOException e) {
            System.out.println(e);
            return;
        }
        finally {
            try {
                sock.close();
            }
            catch (IOException e) {}
        }
    }
}

```

所有这些额外工作只是对 Java 中缺省无符号型数据的功能上的补充，C 程序就不存在这

样的问题。另一种解决方法要简洁一些，涉及到 Java 中的包 `java.math` 中的类 `BigInteger`。在“快速解决方案”一节里将能找到这个方法。

要获得更精确的时间，可以使用更高级的协议——网络时间协议（NTP）。在 RFC1305 以及一些早期的文档里有它的定义，NTP 允许你连向多个时间服务器并持续地更新正确的时间。

3.2 快速解决方案

3.2.1 使用 Echo 协议

Echo 协议是最简单易用的协议之一，常常是执行伪 ping 程序的一个好方法（Java 在 1.4 以前的版本不能直接实现 ping 协议，因为它们不处理 ICMP 协议）。

要使用 echo 服务器，只需在端口 7 上打开套接字（TCP 或者 UDP），发往服务器的任何数据都会返回来，这个协议可用于测试、定时网络操作或者估计 UDP 套接字上的数据丢失。

例如，在程序清单 3.3 中可以看到一个使用 UDP 套接字的 echo 客户端。

注意：因为 UDP 并不确保发送无误，也不保证远方服务器的存在，在程序里设置超时不会因为一个主机一直不应答而永远等待下去了。

3.2.2 编写 TCP Echo 服务程序

写一个 echo 服务器端程序看起来不是那么令人激动，一个简单的 echo 服务器可以做为更复杂的服务器的基础，因为用于连接、接收和传输数据的机制在所有的 echo 服务器里都是一样的。

1. 使用第 2 章里的 `MTServerBase` 对象写一个简洁的 TCP echo 服务器端程序（见清单 3.1）。当然，服务器用的是 7 号端口。

2. 使用下面的代码将你接收的任何内容发回给客户端：

```
while (c!=-1) {  
    c=is.read();  
    if (c!=-1) os.write(c);  
}
```

相关解决方法的位置：

第 2 章 2.2.16 自动处理多线程服务器

3.2.3 编写 UDP Echo 服务程序

Echo 协议的 UDP 服务器与 TCP 服务器稍有些不同，你用不着线程，因为每个包是一个独立的请求。并且，你没有用于将数据送回给客户端的连接，因此不得不从 `DatagramPacket` 对象里选出正确的 IP 地址和端口号。幸运的是，如果使用相同的包发送和接收，就不会有问题，在程序清单 3.2 里可以见到完整的 echo 协议的 UDP 服务器。

程序的 echo 实现部分非常简单:

```
while (true) {  
    try {  
        sock.receive(pack);  
        sock.send(pack);  
    }  
    catch (IOException ioe)  
        System.out.println(ioe);  
}
```

然而, 对不同种类的服务器, 你需要使用 `getAddress` 和 `getPort` 从包里提取 IP 地址和端口号。

3.2.4 合并 TCP 和 UDP 服务程序

如果你在写一个真正的 echo 服务器端程序, 一般需要让它同时监听 TCP 和 UDP 的端口。很多其他类型的服务器端程序都要求有这种双重端口操作。

通常, 将服务器独立工作 (可能使用常用对象来执行处理) 的每一部分取出来比较简单, 那么, 一旦它们都工作正常了, 将它们合并起来就很简单了。典型的, TCP 代码不变, 但要往 `main` 方法里加入代码, 以启动一个新线程, 运行 UDP 服务器。

看看上两节里谈到的两个 echo 服务器程序, 一个用的是 TCP, 另一个用的是 UDP, 要将它们合并在一起, 做法如下:

1. 创建一个新类, 将 UDP 服务器置入线程里, 这个类没有增加任何网络函数; 只是在线程开始运行的时候调用服务器的 `main` 例程。

```
class EchoServerUdp extends Thread {  
    public void run() {  
        UdpEchoServer.main(null);  
    }  
}
```

2. 修改 TCP 服务器程序, 启动线程:

```
public class EchoServer extends TcpEchoServer {  
    public static void main(String [] args) {  
        new EchoServerUdp().start(); // start UDP thread  
        TcpEchoServer.main(args);  
    }  
}
```

使用这个模块化方法可以独立地开发和调度这两个服务器端程序, 如果还有重要的工作要做, 两个服务器对象还可以共享第三个对象来完成这项工作, 因此没必要对真实的逻辑进行重新编码。

3.2.5 使用 Finger 服务

`finger` 协议能让人了解一台机器或者该机器上的用户, 可以向服务器请求常用信息, 即

有关用户或者用户 ID 的清单，或者关于一个特定用户的有关信息。

注意：很多服务器限制使用 finger 或者禁止对它的完全使用，因为它可能代表着安全上的漏洞。特别地，极少数站点允许你得到用户列表，因为它对于黑客试图寻找密码很有用。

打开 finger 服务器的连接，做法如下：

1. 使用 TCP 的 79 端口。
2. 客户端发送一个单行命令请求（由回车加换行符终止），空行表示客户请求的是机器的信息。
3. 服务器返回关于每个用户的信息。

当然，服务器返回的内容依赖于操作系统和使用的 finger 服务器。RFC1288 指出服务器必须至少返回真实的用户名。当然，有些系统将 finger 用于其他目的，因此那并不总是真的。在 Unix 系统里，如果这种服务存在，服务器将返回关于用户、用户计划文件的内容的统计信息。

因为 Java 使用 Unicode 串，有必要将它们转化为 ASCII 字节，不依赖于缺省的编码方案，提供一个从 Unicode 到 ASCII 码的具体映射是一个好方法。还要在请求里加入回车换行符，因为它并不是在所有的操作系统里都是标准的。

假设你将用户列表放在变量 String users 里，将远程机器名放在变量 host 里，下面的代码可以用于编写请求：

```
Socket sock=new Socket(host,79);
OutputStream os = sock.getOutputStream();
InputStream is = sock.getInputStream();
users = users + "\r\n";
os.write(users.getBytes("iso8859_1"));
```

这里使用了 ISO8859_1 编码，也可以使用其他的编码方法，如 ASCII 码或者 CP1252 码。

服务器将对数据作出响应，但是响应的格式不固定，如果在写一个交互式的 finger 客户端程序，在数据到达时，可以简单地将它显示出来。Finger 服务器也支持请求传递，即一个远程计算机在你的请求里“查询”（finger）另一台计算机，然而，极少数站点允许这么做，因为没有更好的理由使用它，并且它意味着要冒安全的风险。

请求的另一个可能选项是/W 标志，如果客户端提供了这个标志，服务器就可以把它解释成一个“想获得详细信息”的请求。但是，服务器并不真正负责提供任何更多的信息或者去改变它的格式。

在清单 3.5 里会看到一个完整的例子。同时，在清单 3.6 里，有一个 JSP 的 finger 网关使用清单 3.5 中的类。

3.2.6 编写 Finger 服务器

编写 finger 服务器要求你接受客户端发送的命令并解析出来，且作出恰当的响应。准确地说，怎么响应将由你来决定。例如，“查询”（finger）nasanews@space.mit.edu 将显示与 NASA

相关的新闻。

要写一个 finger 服务器，需要：

- 寻找/W 参数，即使不想对它作任何处理。含有@的用户 ID 用于请求的传递，绝大多数服务器都不支持。

提示：类 StringTokenizer 对于将接收到的客户端的命令进行解析是很有用的。

- 创建一个从 Unicode 字符到字节的映射。当创建一个 Writer 对象或者包含来自一个 String 对象的字节（用于通过 OutputStream 发送）时，指定一个具体的编码方案（如 ISO8859_1）是一个好方法。

在清单 3.7 里可以见到完整的 finger 服务器端程序。类 InputStreamReader 的方法 readLine 获得命令行。一旦服务器明确了输出，并要发送输出，它就将输出流全部流出，以确保每个字符都发送出去，接着关闭 finger 协议规定中要求的套接字。

相关解决方法的位置：

第 2 章 2.2.16 自动处理多线程服务器

3.2.7 创建一个简单的代理

很多协议，特别是 HTTP 和 FTP 协议，常借助于代理服务器来使用。所谓代理服务器是指一个服务器用来担任客户端的中间人。代理服务器是很有用的，例如在公司网络里，你不想通过客户端（公司网络上）直接访问服务器（公共 Internet 网上）。

在第 12 章里你会看到 HTTP 代理，但你现在就可以根据你知道的内容轻易地建立一个 finger 代理。虽然 finger 代理并不多见，但因为它比较简单，因而比较适合于学习。

代理拥有它传递的底层协议的知识。在这里，finger 代理充当的职责就像是一个 finger 服务器，但它不是直接响应客户端的命令，它代表着另一台主机的职责。

创建一个 finger 代理的步骤如下：

1. 重用清单 3.5 中的 finger 客户端程序，以得到正确输出。
2. 一旦真正的 finger 服务器里出现了输出，就将它中转给原请求者，从客户端的角度来看，finger 代理就是一台 finger 服务器，因为客户端发出请求，finger 代理就响应。事实上，你请求另一台服务器，以得到正确响应并不是很快就很明显的。可以在清单 3.9 中找到这个代理的相关代码。

3. 代理将到来的请求发送到父一级的 finger 服务器，并显示结果：

```
s=fing.finger(parentHost,cmd); // ask parent
os.write(s,0,s.length()); // send output
os.flush(); // flush the output
```

相关解决方法的位置：

第 13 章 编写一个代理服务器程序

3.2.8 使用 Whois

Whois 协议与 finger 类似，但它向主服务器查询很多不同类型的问题。主 whois 数据库在 whois.internic.net。网络解决方案，原始的注册登记员，在 whois.networksolutions.com 有一个服务器。这个协议（由 RFC954 定义）与 finger 类似，在 43 端口打开 TCP 套接字，发送单行命令，并以回车换行符终止。

服务器到底要查找什么内容依赖于服务器。你也可以使用服务器指定的命令集来从几个方面限制查询（见表 3.1）。如果登录进服务器（端口 43），并发出 help 命令，将很可能看到可接受的命令清单以及有关服务器的其他信息。

清单 3.10 里显示的是一个能查询任何 whois 服务器的客户端程序。因为命令的格式随服务器的不同而不同，程序简单地发送你键入命令行里的内容，并打印出服务器返回的内容。

3.2.9 查询对人可读格式的时间

支持 RFC867 的机器允许你获取对人可读格式的时间，在端口 13 上使用 TCP 或者 UDP。要查询时间，步骤如下：

1. 对 TCP 协议，简单地创建一个连接，服务器会将时间值返回（无论它像是什么格式），并关闭这个连接。
2. 对 UDP 协议，发出一个数据报，虽然数据报的内容不重要。

在表 3.2 中可以看到一些流行的时间服务器的列表。这些服务器有很多严格按照原子时钟来设置，提供非常准确的时间。当然，网络延迟以及其他因素会使你读到的时间值有一位误差，虽然最复杂精密的服务器竭力为解决这个问题而负责。

下面有一小段代码，主要是使用 TCP 协议读来自于美国海军气象台的时间值：

```
try {
    int c;
    Socket sock=new Socket("tock.usno.navy.mil",13);
    InputStream is = sock.getInputStream();
    do {
        c=is.read();
        if (c!=-1) System.out.print((char)c);
    } while (c!=-1);
    }
    catch (IOException e) { System.out.println(e); }
```

3.2.10 对 NIST 时间串进行解码

NIST 有为返回的时间串值定义好的格式，其格式如下：

JJJJJ YR-MO-DA HH:MM:SS TT L H msADV UTC(NIST) *

- JJJJJ——指修正的 Julian 日，修正意味着它是真正的 Julian 日的后 5 位，按规定，

开始于公元前 4713 年的 1 月 1 日，这个数加上 2,400,000 就变成了 Julian 日了。

- YR-MO-DA——这是年，月，日的日期格式。
- HH:MM:SS——时，分，秒格式的时间，通常是 UTC 时间。
- TT——这段代码表明美国是否符合夏令时。如果值为 00，夏令时无效，如果为 50，夏时间有效，如果当前日期是在一个月内的夏令时变化范围内，其值将随之减少。
- L——这个字符表明这个月末的时钟是否要进行闰秒纠正。当这个域值为 1 时，表明将有一个闰秒加到月末时钟里；值为 2 时，表明要减去一个闰秒；值为 0 时，表明这个月没有闰秒。
- H——这个字符表明服务器当前的“健康”状态。值为 0 表明服务器操作正常；值为 1，表明时间误差在 5 秒范围内；值为 2 表明可能有超过 5 秒的误差；值为 4 表明什么地方出了问题，时间可能整个出了问题。
- msADV——这是为减轻网络延迟加到时间值上的毫秒数。
- UTC (NIST)——这个时区指示器指的是同等通用时间，在过去，UTC 以格林威治 (GMT) 时间或者祖鲁 (zulu) 时间而出名。
- *——这是一个准时时间标志，当收到这个字符时，表明包里的时间是正确的。当然，由于网络延迟以及其他问题，可能有一位的误差，但这样会导致连接关闭。

3.2.11 查询机器可读格式中的时间

查询机器可读格式的时间，常用方法如下：

1. 读字节，并减去 2,208,988,800（从 1900 到 1970 年这间的秒数，考虑了闰年）。
2. 乘以 1000 转换成毫秒数。
3. 调用 Date 对象的 setTime 方法获得正确的日期和时间。

清单 3.15 与清单 3.14 一样，实现了上述方法，虽然清单 3.14 使用了位操作来克服大数运算问题，即对在 C 语言中的类似处理模型化的一种策略。清单 3.15 使用了类 BigInteger，更像是一个 Java 方法，这个类用于处理比 Java 常用数据类型要大的数。

程序使用方法 read 的特殊形式来取到来数据的 5 字节数组的后 4 个字节。接着，将第一个字节设为 0，用这个方法，类 BigInteger 的构造函数将它接收为 5 字节的正整数。如果你试图使用 4 字节的数组，构造函数将创建一个负整数。

当然，一旦这个数是 BigInteger 格式的，就必须使用 BigInteger 进行其他操作，到了取 Date 对象的时候，就可以调用 longValue 来从 BigInteger 对象那里得到 long 型值。

清单 3.15 这个 RFC868 客户端使用类 BigInteger 处理到来的时间

```
import java.util.*;
import java.net.*;
import java.io.*;
import java.math.*;
```



```
import java.text.DateFormat;

public class Client868big {
    public static void main(String [] args) {
        Socket sock=null;
        try {
            sock=new Socket(args[0],37);
            byte [] inarray=new byte[5];
            Date date=new Date();
            InputStream is=sock.getInputStream();
            is.read(inarray,1,4);
            BigInteger tick;
            inarray[0]=0; // make sure number is positive
            tick = new BigInteger(inarray);
            //number of second between 1900 and 1970
            // convert between 1900/1970
            tick=tick.subtract(new BigInteger("2208988800"));
            // convert to ms
            tick=tick.multiply(new BigInteger("1000"));
            date.setTime(tick.longValue());
            DateFormat df=DateFormat.getDateTimeInstance(
                DateFormat.FULL,DateFormat.FULL);
            System.out.println(df.format(date));
        }
        catch (IOException e) {
            System.out.println(e);
            return;
        }
        finally {
            try {
                sock.close();
            }
            catch (IOException e) {}
        }
    }
}
```

3.2.12 编写时间服务程序

你可以写一个时间服务器来处理到来的对机器可读时间或者对人可读时间的请求，简单做法如下：

1. 注意到来的连接（或者数据报）。
2. 以合适格式的时间响应。
3. 一旦发送完数据（如果使用的是 TCP 协议的服务），就关闭连接。

在清单 3.12 里可以找到 RFC867 的 TCP 服务器的一个示例，因为 `DateFormat` 对象可以创建你喜欢的任何格式，你可以用它产生你为客户端服务而用的串，这个串必须以回车换行

符结尾。

清单 3.13 里也有一个 RFC868 服务器，使用的也是 TCP，其使用 `getTime` 方法检索当前日期和时间，变成自 1970 年 1 月 1 日起的毫秒数，要写这个服务器端程序，做法如下：

1. 将那个数转换成秒（除以 1000）。
2. 加上从 1900 到 1970 年之间的秒数（算上闰年，但不算闰秒，大约为 2,208,988,800）。
3. 将每个字节提取到字节数组中去，发送到输出套接字里边。使用标准的位操作（&和 >> 操作）很容易做。

下面是典型时间服务器的一部分，将结果拆成一个个字节：

```
long tick = new Date().getTime()/1000;
byte [] outarray = new byte[4];
tick+=2208988800L;
outarray[0]=(byte)((tick & 0xFF000000L)>>24);
outarray[1]=(byte)((tick & 0xFF0000L)>>16);
outarray[2]=(byte)((tick & 0xFF00L)>>8);
outarray[3]=(byte)(tick & 0xFFL);
os.write(outarray);
os.flush();
```

3.2.13 选用 Unicode 作字节映射

Java 使用 Unicode 字符，它用 16 个二进制位表示字符，因为网络通常以字节的形式传输，所以有必要将 Unicode 字符转换成字节，当往 `OutputWriter` 里写且没有指定编码时，系统将使用缺省编码。但是，对网络协议而言，最好是为自己选择一种编码。

指定编码有下述几种方法：

- 对于 `OutputStreamWriter`，可以在构造函数里指定编码。
- 如果直接使用字节工作，可以使用指定的编码来调用 `String` 对象的方法 `getBytes`。

编码方法是指系统试图匹配的串。所有 Java 的安装都支持一些基本的编码，尽管较新版的 Java 可能支持较老的版本。文件 `i18n.jar` 增加了对很多其他编码的支持（见表 3.3）。

记住所有常规可打印的 ASCII 字符本身都是 Unicode 字符，因此映射通常不难。问题在于当使用了出现在其他语言中的字符或者奇怪的情况（例如，PC 中的特殊字符）。UTF 提供了通过 8 位通道来对 Unicode 字符编码的特殊方法。但是，除非接收者期望 UTF，一般情况下，这种格式并没有太大意义。

表 3.3 Java 认可的编码方式

名称	编码
ASCII	美国信息交换标准码
Cp1252	Windows Latin-1
ISO8859_1	ISO 8859-1, Latin alphabet No. 1

续表

名称	编码
UnicodeBig	16-bit UTF, big-endian byte order, with byte-order mark
UnicodeBigUnmarked	16-bit UTF, big-endian byte order
UnicodeLittle	16-bit UTF, little-endian byte order, with byte-order mark
UnicodeLittleUnmarked	16-bit UTF, little-endian byte order
UTF8	8-bit UTF
UTF-16	16-bit UTF, byte order specified by a mandatory initial byte-order mark
Big5	Big5, Traditional Chinese
Cp037	USA, Canada (Bilingual, French), Netherlands, Portugal, Brazil, Australia
Cp273	IBM Austria, Germany
Cp277	IBM Denmark, Norway
Cp278	IBM Finland, Sweden
Cp280	IBM Italy
Cp284	IBM Catalan/Spain, Spanish Latin America
Cp285	IBM United Kingdom, Ireland
Cp297	IBM France
Cp420	IBM Arabic
Cp424	IBM Hebrew
Cp437	MS-DOS United States, Australia, New Zealand, South Africa
Cp500	EBCDIC 500V1
Cp737	PC Greek
Cp775	PC Baltic
Cp838	IBM Thailand extended SBCS
Cp850	MS-DOS Latin-1
Cp852	MS-DOS Latin-2
Cp855	IBM Cyrillic
Cp856	IBM Hebrew
Cp857	IBM Turkish
Cp858	Variant of Cp850 with Euro character

续表

名称	编码
Cp860	MS-DOS Portuguese
Cp861	MS-DOS Icelandic
Cp862	PC Hebrew
Cp863	MS-DOS Canadian French
Cp864	PC Arabic
Cp865	MS-DOS Nordic
Cp866	MS-DOS Russian
Cp868	MS-DOS Pakistan
Cp869	IBM Modern Greek
Cp870	IBM Multilingual Latin-2
Cp871	IBM Iceland
Cp874	IBM Thai
Cp875	IBM Greek
Cp918	IBM Pakistan (Urdu)
Cp921	IBM Latvia, Lithuania (AIX, DOS)
Cp922	IBM Estonia (AIX, DOS)
Cp930	Japanese Katakana-Kanji mixed with 4370 UDC, superset of 5026
Cp933	Korean Mixed with 1880 UDC, superset of 5029
Cp935	Simplified Chinese Host mixed with 1880 UDC, superset of 5031
Cp937	Traditional Chinese Host mixed with 6204 UDC, superset of 5033
Cp939	Japanese Latin Kanji mixed with 4370 UDC, superset of 5035
Cp942	IBM OS/2 Japanese, superset of Cp932
Cp942C	Variant of Cp942
Cp943	IBM OS/2 Japanese, superset of Cp932 and Shift-JIS
Cp943C	Variant of Cp943
Cp948	OS/2 Chinese (Taiwan), superset of 938
Cp949	PC Korean
Cp949C	Variant of Cp949
Cp950	PC Chinese (Hong Kong, Taiwan)

续表

名称	编码
Cp964 Cp970	AIX Chinese (Taiwan) AIX Korean
Cp1006	IBM AIX Pakistan (Urdu)
Cp1025	IBM Multilingual Cyrillic: Bulgaria, Bosnia, Herzegovina, Macedonia (FYR)
Cp1026	IBM Latin-5, Turkey
Cp1046	IBM Arabic—Windows
Cp1097	IBM Iran (Farsi) /Persian
Cp1098	IBM Iran (Farsi) /Persian (PC)
Cp1112	IBM Latvia, Lithuania
Cp1122	IBM Estonia
Cp1123	IBM Ukraine
Cp1124	IBM AIX Ukraine
Cp1140	Variant of Cp037 with Euro character
Cp1141	Variant of Cp273 with Euro character
Cp1142	Variant of Cp277 with Euro character
Cp1143	Variant of Cp278 with Euro character
Cp1144	Variant of Cp280 with Euro character
Cp1145	Variant of Cp284 with Euro character
Cp1146	Variant of Cp285 with Euro character
Cp1147	Variant of Cp297 with Euro character
Cp1148	Variant of Cp500 with Euro character
Cp1149	Variant of Cp871 with Euro character
Cp1250	Windows Eastern European
Cp1251	Windows Cyrillic
Cp1253	Windows Greek
Cp1254	Windows Turkish
Cp1255	Windows Hebrew
Cp1256	Windows Arabic

续表

名称	编码
Cp1257	Windows Baltic
Cp1258	Windows Vietnamese
Cp1381	IBM OS/2, DOS People's Republic of China (PRC)
Cp1383	IBM AIX People's Republic of China (PRC)
Cp33722	IBM-eucJP—Japanese, superset of 5050
EUC_CN	GB2312, EUC encoding, Simplified Chinese
EUC_JP	JIS X 0201, 0208, 0212, EUC encoding, Japanese
EUC_KR	KS C 5601, EUC encoding, Korean
EUC_TW	CNS11643 (Plane 1-3), EUC encoding, Traditional Chinese
GBK	GBK, Simplified Chinese
ISO2022CN	ISO 2022 CN, Chinese (conversion to Unicode only)
ISO2022CN_CNS	CNS 11643 in ISO 2022 CN form, Traditional Chinese (conversion from Unicode only)
ISO2022CN_GB	GB 2312 in ISO 2022 CN form, Simplified Chinese (conversion)
ISO2022JP	JIS X 0201, 0208 in ISO 2022 form, Japanese
ISO2022KR	ISO 2022 KR, Korean
ISO8859_2	ISO 8859-2, Latin alphabet No. 2
ISO8859_3	ISO 8859-3, Latin alphabet No. 3
ISO8859_4	ISO 8859-4, Latin alphabet No. 4
ISO8859_5	ISO 8859-5, Latin/Cyrillic alphabet
ISO8859_6	ISO 8859-6, Latin/Arabic alphabet
ISO8859_7	ISO 8859-7, Latin/Greek alphabet
ISO8859_8	ISO 8859-8, Latin/Hebrew alphabet
ISO8859_9	ISO 8859-9, Latin alphabet No. 5
ISO8859_13	ISO 8859-13, Latin alphabet No. 7
ISO8859_15_FDIS	ISO 8859-15, Latin alphabet No. 9
JIS0201	JIS X 0201, Japanese
JIS0208	JIS X 0208, Japanese
JIS0212	JIS X 0212, Japanese

续表

名称	编码
JISAutoDetect	Detects and converts from Shift-JIS, EUC-JP, ISO 2022 JP (conversion to Unicode only)
Johab	Johab, Korean
KOI8_R	KOI8-R, Russian
MS874	Windows Thai
MS932	Windows Japanese
MS936	Windows Simplified Chinese
MS949	Windows Korean
MS950	Windows Traditional Chinese
MacArabic	Macintosh Arabic
MacCentralEurope	Macintosh Latin-2
MacCroatian	Macintosh Croatian
MacCyrillic	Macintosh Cyrillic
MacDingbat	Macintosh Dingbat
MacGreek	Macintosh Greek
MacHebrew	Macintosh Hebrew
MacIceland	Macintosh Iceland
MacRoman	Macintosh Roman
MacRomania	Macintosh Romania
MacSymbol	Macintosh Symbol
MacThai	Macintosh Thai
MacTurkish	Macintosh Turkish
MacUkraine	Macintosh Ukraine
SJIS	Shift-JIS, Japanese
TIS620	TIS620, Thai

第4章 TFTP 协议

4.1 深入介绍

当我听有人谈到要是生活在古代那该有多精彩的时候，总是感到很吃惊。我并不是指那些渴望年轻的人们，而是指那些想要生活在过去的南部地区，或者荒野的西部地区，甚至是中世纪的欧洲。那些认为回到过去是好事的人可能不知道在那种年代里发生了些什么事情。

每年本地公园都有一个展览，向人们显示德克萨斯的这个地区的居民早期居住时的情景。你将为早期生活中有如此多的工作感到吃惊，人们不得不自己制造肥皂，自己纺织布匹和缝制衣服，还要为每一口食物而劳作生产，那是一项全职工作。在某种程度上，女士们甚至发现时间可以让自己的饰带成为奢侈品。如果我不得不自己制作肥皂，那我就领先了。

现在，我们绝大多数人不用自己生产自己的食物、肥皂、衣服以及在沃尔玛商店可以廉价得到的其他物品。即使你种植自己的食物，你可能要买已制成的化肥、包装好的种子，还需要功率足够的工具来帮助你。

尽管很多爱好技术的人组装自己的 PC，那也是很久远的事了，我们大多数人可以将主板放到已经制好的主机箱里。但我们有多少人能够制作主板的每一部分，将它们恰当地焊接，再将我们手工制作的这个物品放到自己制作的容器（在自己的后院作坊里精炼而成）里边呢？我猜测没有一个人会举起手来说会。

在上一章里，使用的协议都非常简单。其中的绝大多数协议程序都可以使用 Telnet 来进行测试，事实上就是如此简单。但是，实际应用的协议往往比那些协议要复杂得多。你可能不会真正要写出处理这些更复杂的协议所要求的每行代码，就像你不会试图去制作自己的奔腾 IV 处理器一样。

在这一章里，将要研究一个相对简单的协议，但它比第 3 章的协议要复杂得多。为简化学习，你将会发现当碰到复杂的协议的时候，开放源码软件可以简化你的工作。

特别地，这一章讨论的是一般文件传输协议（TFTP），它由 RFC1350 文档定义。你考虑文件传输的时候，很可能想到的是 FTP（文件传输协议）。由名字可以看出，FTP 比 TFTP 要复杂一些（第 6 章会讨论到 FTP）。

许多无盘工作站使用 TFTP 来载入它们需要的来自服务器的文件。当一台计算机需要从另一台计算机请求得到一个文件并且无需登录到一台远程机器的时候，使用 TFTP 将很合适。典型地，一台 TFTP 服务器只提供有限数量的文件，任何需要这些文件的人都可以使用该服务来访问这些文件。

4.1.1 关于 TFTP 协议

TFTP 协议特地设计成易处理的形式，它使用了用户数据报协议（UDP）套接字，效率比较高，但这也意味着 TFTP 要为数据传输的不可靠负责。

TFTP 服务器在 69 端口上监听到来的数据包，客户端简单地使用一个随机的端口号。初始的包交换允许服务器将客户端移到一个不同的端口号上。

每个数据报有一个 opcode（16 位字），表示包的类型，包的类型共有五种（见表 4.1）。协议简单的一个原因是包的数目可能比较小。但是，另外几个特征使得编写 TFTP 软件比较容易：

- 发送的每个包引起一个响应。在收到这个响应之前，不能继续其他的工作，如果超时，可以重发数据包。因此，每次都要记住一个包中的数据（将它们暂存）。
- 每个数据包都有 512 字节的文件。更小的数据包要指出文件的终止处。
- 几乎所有的错误都引起传输的终止。服务器端返回错误码和相关选项的描述，并终止这个事务。
- 协议只承认三种文件类型。其中有一种文件类型已经没有用了，所以实际使用当中，只需注意另两种文件类型。

表 4.1 TFTP 数据包类型

码值	包名	说明
1	RRQ	请求读(请求获得文件)
2	WRQ	请求写(请求发送文件)
3	DAT	文件数据
4	ACK	确认，继续进行传输
5	ERR	发生错误

当一台计算机要从另一台计算机读一个文件，它就向远程计算机的 69 端口上发送读请求（RRQ），要写文件，就发送写请求（WRQ）。数据包里都有类型码（也即 opcode）、文件名（由 0 字节终止）和模式串（也由 0 字节终止）。文本文件的模式是 netascii（网络 ASCII），而二进制文件的模式是 8 位的二进制位组（octet）。第三种法定方式——邮件，现在已经废止不用了。

读一个文件的时候，远程计算机将以一个数据包作响应。写请求会产生一个 ACK 包，远程计算机将在 69 以外的端口发出它的响应。客户端将使用这个端口用作后来的数据传输。

数据包内有 2 字节的块号（块号从 0 开始）。通常，512 字节的数据将以块号来记，然而，可能有更少的字节出现（包括 0 个字节）。短的块要指示文件的终止位置。块号只有 2 字节长，每块是 512 字节，所以 TFTP 不会处理大于 32M 字节的文件。

一个 ACK 包也有它的块号。如果 ACK 是针对 WRQ 包，其块号将会是 0；否则，它的块号要对应于最后一个数据包的块号。

不管在什么时候，每台机器都可以发送一个 ERR 包。它包含一个两字节的错误码值（见表 4.2），以及一个空终止符的错误串。例如，当服务器要拒绝一个连接的请求时，它就可以发送一个 ERR 包。

表 4.2 错误类型

码值	说明
0	未定义
1	文件未找到
2	非法访问
3	磁盘满
4	非法 TFTP 操作
5	未知传输 ID（端口号）
6	文件已经存在
7	无此用户

ERR 包不用确认，但有可能 ERR 的发送者要等待一小段时间，看是否继续从其他计算机那儿接收数据，在这种情况下，可以假设其他计算机没有接收错误消息，应该将错误消息重发。然而这种要求并不是必需的。

实际上，要求有某种超时和重试次数机制。假设要发送数据包 100 到接收者，等待确认（ACK）包，但在预定时间内没有到达，可以将数据包重发，当然，接收者这时可能已经读到了上次发送的数据包，在这种情况下，ACK 包应该丢失。接收者简单地丢弃重复发送的数据包，并重新发回一个 ACK 包。也可以假设数据包没到就意味着 ACK 包丢失，可以重发数据包来完成传输。

实际上，大致意思是这个事务（数据传输）的两边必须都实现超时机制，并准备接收重复的数据包，每次只有一个包是突出的，因而容易实现。

注意：不要错误地认为只有 ACK 包表示确认。事实上，两边计算机发送的每个包都期待另一个包作响应。这样，对发一个已经发送的数据包而言，应该接收一个 ACK 包作响应。但是从另一方的角度来看，一个 ACK 包会引起另一个数据包的到来。

不会产生响应的唯一的 ACK 包是最后一个 ACK 包（对应于一个小于 512 字节的数据包）。一旦接收者发送了最后一个 ACK 包，就可以关闭连接了。但是，可能要等待一段时间看看发送方是否重发那个数据包，如果重发，可以假设最后一个 ACK 包丢失，将它重发。这种做法是可选的，却是一个好方法。

4.1.2 Play by Play

看看一个典型的 TFTP 事务，如表 4.3 所示。计算机 A 是一个工作站，要从计算机 B 那儿读取一个文件。计算机 B 运行着 TFTP 服务器端程序，使用的是常规端口 69。

表 4.3 计算机 A 读来自 B 计算机的数据

计算机 A		计算机 B
数据	到远程端口	响应
RRQ	69	DATA
ACK	3242	DATA
ACK	3242	DATA
ACK	3242	

在这个例子里面，服务器已经将客户端移到 3242 端口（随机选择）上，让 69 端口为更多的 RRQ（或者 WRQ）请求开放，文件大小必须介于 1,024 到 1,535 字节之间。如果文件长度为 1,024 字节，最后一个数据包里将没有数据，以简单的文件结束。如果文件为 1,536 字节，最后一个数据包将包含 512 字节数据，接收者将期待另一个数据包，即使它是 0 长度的包。因此，文件长度至少为 1,024 字节，但不能超过 1,535 字节。

如果计算机 A 在向服务器写数据，这个事务将如表 4.4 所示。

表 4.4 计算机 A 向计算机 B 写数据

计算机 A		计算机 B
数据	到远程端口	响应
WRQ	69	ACK
DATA	3242	ACK
DATA	3242	ACK

在此例当中，文件长度必须介于 512 字节和 1,023 字节之间。如果有一边发觉为一个响应等待时间太长，可以重发它的数据。因为在任一给定的时间内，只可能有一个包是重要的，这将易于编写程序。

4.1.3 TFTP 客户端应用

如果对协议比较熟悉，可以编写简单的客户端类。客户端可以发送或者接收文件，因为它不必处理多个请求，没有任何理由要处理多线程和套接字。

在本章后边的“快速解决方案”一节里，将会看到一个完整的 TFTP 客户端程序（见清

单 4.5)。那个类 (TftpSocket) 从类 DatagramSocket 继承而来, 它工作得很好, 尽管你可以将这个类改成封装一个套接字的形式。

新的类有构造函数, 映射成 DatagramSocket 可得到的构造函数, 然而 TftpSocket 类的逻辑要求都是一样的。这样, create 这个私有函数要注意所有的构造函数。

TftpSocket 构造函数除了需要常规的 DatagramSocket 构造函数所要求的参数外, 还需要文件名、文件类型 (netascii 或者 octet) 以及网络主机名等参数。尽管遇到 netascii 文件时, 试图处理行结束问题比较容易, 客户端还是没有作这种尝试。

构造函数并没有真正开始一个事务, 它只是将主机名解析成 InetAddress 对象, 将串参数转换成字节数据, 因为服务器端要的是字节数据, 而不是 Unicode 串。

一旦构造了 TftpSocket 对象, 将可以调用 fsend 或者 freceive 来执行你想要的操作, 先看看要接收一个文件的情形。

所有的请求和接收一个文件的逻辑要求都在 freceive 方法里, 但是, 其细节都隐藏在两个专门的子例程里: sendPacket 和 rcvPacket。这两个例程包含有数据包事务的所有专门知识, 而 freceive 方法控制着数据包的次序和超时逻辑。在清单 4.1 中可以看到 freceive 方法。

清单 4.1 freceive 方法通过 TFTP 接收一个文件

```
// Get a file
public boolean freceive() throws SocketException {
    boolean rv;
    int n;
    try {
        fileout=new FileOutputStream(fnstring);
    }
    catch (Exception e) { return false; }
    try {
        int retry=0;
        block=1;
        rv=sendPacket(RRQ);
        if (!rv) return rv;
        while (!EOF) {
            do {
                n=rcvPacket();
                if (n==TIMEOUT) {
                    if (retry++>5) return false;
                    continue;
                }
                retry=0; // reset retry count
                if (n==ERR) return false;
                if (n==UNKNOWN) {
                    sendErrPacket(0,"Unknown error");
                    return false;
                }
            }
        }
    }
}
```

```

        } while (n!=DAT);
    }
    return true;
}
finally { // close in all cases
    if (!streamClose(fileout)) return false;
}
}

```

在有错误包的情况下，`sendPacket` 例程会产生常见错误。然而，你也可以调用 `sendErrorPacket` 来创建一个特定的错误。调用方法 `sendPacket` 将创建对当前数据块的确认。但是，如果服务器端重发数据块，可能需要重复前一个确认，这样方法 `sendAck` 将创建一个具体的确认包。

`freceive` 方法必须创建一个 `FileOutputStream`，`rcvPacket` 用它来写一个输出文件。当然，不管出现什么情况，这个文件必须要关闭。因此 `freceive` 方法实现了一个 `finally` 语句，以确保即使有错的情况下，程序也会关闭这个流 (`FileOutputStream`)。

`fsend` 方法与 `freceive` 方法非常类似（见清单 4.2）。它打开一个 `FileInputStream` 对象，当然，在这种情况下程序是在读一个现存的文件。另外，`sendPacket` 和 `rcvPacket` 方法处理所有的底层细节部分。

清单 4.2 本方法发送一个文件

```

// Send a file
public boolean fsend() throws SocketException {
    boolean rv;
    try {
        filein = new FileInputStream(fnstring);
    }
    catch (Exception e) { return false; }
    try {
        int n;
        int retry=0;
        block=0;
        rv=sendPacket(WRQ);
        if (!rv) return rv;
        while (!EOF) {
            do {
                n=rcvPacket();
                if (n==TIMEOUT) {
                    if (++retry>5) return false;
                    if (block!=0)
                        if (!sendPacket(DAT)) return false;
                    continue;
                }
                if (n==ERR) return false;
                if (n==UNKNOWN) {

```

```

        sendErrPacket(0, "Unknown error");
        return false;
    }
    } while (n != ACK);
    if (!sendPacket(DAT)) return false;
    }
    return true;
}

finally {
    streamClose(filein); // close no matter what
}
}

```

`rcvPacket` 方法返回接收到的包的类型，但调用的例程需要知道更多的细节。因此，定义了一些私有的返回代码的类没有与包的代码号重迭，特别是下面这些返回代码：

- **NONE**——包接收不希望接收的那个包。例如，从这个可以指出一个包的重复确认。
- **UNKNOWN**——包的次序发生错误。例如，服务器端对客户端还没发送的一个包发送了确认信息。
- **TIMEOUT**——指出发生了超时。

`TftpSocket` 类有一个用于测试的 `main` 函数。如果不提供参数，这个类将以服务器的形式启动。然而，可以为它提供文件名、文件类型（`netascii` 或者 `octet`）以及主机名作参数。如果要发送文件，还可以用 `S` 作最后一个参数（缺少情况下是接收文件）。

单步执行这段代码将很有指导意义，这个类有几个实例变量客户端需要使用，包括如下这些：

- **String fnstring**——这是目标文件名，是一个 Java 串。
- **byte[] fn**——客户端也需要文件名，是一个字节数组。
- **byte[] ftype**——当指定文件类型的时候，客户端将它存成字节数组的形式。
- **InetAddress hostaddr**——指远程计算机（这里是指服务器端）。
- **int port**——使用的端口号，初始的时候它是 69，但当服务器响应时，它将使用另一个端口号。
- **int block**——客户端使用这个变量跟踪它期望接收的数据块。
- **FileInputStream filein**——当发送文件时使用这个流。
- **FileOutputStream fileout**——当接收文件时使用这个流。
- **boolean EOF**——文件操作结束时，这个标志的值将为真。

接收文件的时候，客户端等待一个特殊的块，由变量 `block` 决定。但是，有可能因为“网络数据丢失”，服务器端可能重发客户端已处理过的数据块，客户端必须重新确认那些旧数据包，这样服务器才继续发送新的数据包。下列代码将做这些工作：

```

case DAT:
    // check block # and store
    blockin = (byt[2] << 8) + byt[3];

```

```

    if (blockin>block) return UNKNOWN; // higher than expected
    if (blockin<block) {                // packet re-sent
        sendAck(blockin); // re-send ACK
        return NONE;
    }
    // block number matches
    try {
        fileout.write(byt,4,pack.getLength()-4);
    }
    catch (IOException e) { }
    // DAT packet <516 bytes means end of file
    if (pack.getLength()<516) EOF=true;
    sendPacket(ACK); // ack current
    block++;
    return DAT;

```

注意到达的一个块其块号大于期待值，就会产生错误。因为服务器端直到客户端确认了最后一个发送的数据包才能发送新的数据包，这个错误应该不会发生。

4.1.4 TFTP 服务器端应用

TFTP 服务器端和客户端的区别是什么？区别很小。当然，服务器端监听 69 端口，也为处理每个客户端请求分配一个端口。否则，处理的事务将是相同的。服务器端响应 RRQ 和 WRQ 请求，而不是产生它们，一旦客户端发出其中的一个请求，剩下的事务就与客户端的事务一样了。也就是说，在这以后，服务器端或者发送 ACK 包以响应到来的数据包，或者必须发送数据包，并等待 ACK 包的到来。

因为服务器端与客户端是如此类似，可以试着使用相同的类来完成两者之间的对话，这样比较有意义，这个类提供了静态成员函数 (tftpd)，来处理服务器逻辑（见清单 4.3），这个函数会阻塞，这样将可能从一个线程里调用它。

清单 4.3 本方法充当一个服务器，响应 WRQ 和 RRQ 包

```

public static void tftpd() {
    DatagramSocket sskt;
    try {
        sskt = new DatagramSocket(69);
    }
    catch (SocketException e)
        System.out.println("Unable to start server: " + e);
    return;
}
byte [] buf = new byte[1024];
DatagramPacket pkt=new DatagramPacket(buf,1024);
while (true) {
    try {
        // wait for WRQ or RRQ -- all others are errors

```

```

sskt.setSoTimeout(0);
sskt.receive(pkt);
int port=pkt.getPort();
byte[] byt=pkt.getData();
int code=(byt[0]<<8)+byt[1];
if (code!=WRQ && code!=RRQ) {
    byte[] errary = new byte[4];
    errary[0]=(byte)((ERR>>8)&0xFF);
    errary[1]=(byte)(ERR & 0xFF);
    errary[2]=(byte)0;
    errary[3]=(byte)0;
    errary[4]=0;
    pkt.setData(errary);
    sskt.send(pkt);
    continue;
}
// Start new thread (use private constructor)
TftpSocket worker=new TftpSocket();
int zlen,zlen1;
worker.port=pkt.getPort();
worker.hostaddr=pkt.getAddress();
for (zlen=0;byt[zlen+2]!=0;zlen++);
worker.fnstring=new String(byt,2,zlen);
worker.fn=worker.fnstring.getBytes();
for (zlen1=0;byt[zlen+zlen1+2]!=0;zlen1++);
worker.ftype=new String(byt,zlen+2,zlen1).getBytes();
worker.serverCode=code;
new Thread(worker).start();
}
catch (IOException e) { /* ignore */ }
}
}

```

当使用客户端的时候，套接字提供了与常规的 `DatagramSocket` 构造函数相对应的构造函数。这些构造函数中的一个主函数是将 `String` 参数转换成字节数组。但是，作为服务器端，文件名和来自客户端的其他信息已经是字节数组，因此没必要执行这些转换。基于上述原因，`TftpSocket` 类提供了私有的缺省构造函数，允许 `tftpd` 函数自己填充必需的数据。因为构造函数是私有的，只有 `TftpSocket` 类的成员可以使用它。

服务器端等待到来的 `RRQ` 或者 `WRQ` 包，当接收到其中之一时，就启动一个基于 `TftpSocket` 对象的新线程（那就是为何 `TftpSocket` 要实现 `Runnable` 接口）。`Run` 方法只有在 `TftpSocket` 的特定实例为客户端服务时才运行（见清单 4.4）。

服务器线程与客户端都使用 `rcvPacket` 和 `sendPacket` 例程。当然，在服务器端激活的文件流与客户端是相反的，客户端使用 `FileInputStream`，那么服务器端就使用 `FileOutputStream`，反之亦然。这样才有意义，因为有一边在读，那么另一边必须也在写。

清单 4.4 TFTP 服务器线程为客户端服务

```

public void run() { // this is the server "thread"
    if (serverCode==WRQ) {
        // Set up output stream
        try {
            fileout=new FileOutputStream(fnstring);
        }
        catch (IOException e) {
            sendErrPacket(2,"Access violation");
            return;
        }
        // send ACK0
        sendAck(0);
        // wait for DATA
        int n;
        do {
            try {
                n=rcvPacket();
            }
            catch (SocketException e) { n=ERR; }
            if (n==ERR)
                streamClose(fileout);
            return;
        }
        if (n!=DAT&& n!=NONE) {
            sendErrPacket(0,"?");
            streamClose(fileout);
            return;
        }
    } while (!EOF);
}
if (serverCode==RRQ) {
    // set up input stream
    try {
        file_in=new FileInputStream(fnstring);
    }
    catch (IOException e) {
        sendErrPacket(1,"File not found");
        return;
    }
    // send DATA
    block=1;
    int retry=0;
    do {
        int n;
        if (!sendPacket(DAT)) {
            sendErrPacket(0,"?");

```

```

        streamClose(filein);
        return;
    }
    do {
        try {
            n=rcvPacket();
        }
        catch (SocketException e) { n=ERR; }
        if (n==ERR) {
            streamClose(filein);
            return;
        }
        if (n==UNKNOWN||n==RRQ||n==WRQ||n==DAT) {
            sendErrPacket(4,"Illegal");
            streamClose(filein);
            return;
        }
        if (n==TIMEOUT && ++retry>5) {
            sendErrPacket(0,"Timeout");
            streamClose(filein);
            return;
        }
        if (n==ACK) {
            block++;
            retry=0;
        }
    } while (n!=ACK&& n!=TIMEOUT);
} while (!EOF);
}
}

```

4.1.5 更简单的一种方法

所有的工作都是创建 TFTP 服务器和 TFTP 客户端吗？是的，TFTP 是在 Internet 上较简单的协议之一，但是，如果你仔细搜索，可能会在网上找到一些帮助。

特别地，GNU 工程有一个 TFTP 类，是由 Mark Benvenuto 编写的，网址是 <http://www.gnu.org/software/java/java-software.html>。这里的类与本章给出的代码中的类的工作方式比较类似，尽管细节有些不同。

包 `gnu.inet.tftp` 包含多个有用的类（当然都给出了源码）：

- `Tftp`——这个简单的类不过是一个 `main` 函数，打印一个帮助消息（用法），还有一个构造函数。构造函数创建一个 `TftpConnection` 完成所有的工作。这个类实质上是类 `TftpConnection` 的用户接口，并形成可用的命令行式的客户端程序。
- `TftpConnection`——在这个类中，可以找到处理客户端连接的代码。
- `TftpLogStream`——GNU 的 TFTP 服务器能够创建日志文件，并以消息信息的形式显

示出来，这个类用于处理那些日志。

- `TftpReadConnection`——这个类用于管理对一个文件的读操作。
- `TftpServerConnection`——这个类代表着 TFTP 连接的服务器方。
- `TftpWriteConnection`——这个类用于管理对一个文件的写操作。
- `Tftpd`——这是为 `TftpServerConnection` 实现的一个用户接口类。
- `LineBufferedOutputStream`——这是由 TFTP 类使用的用途广泛的缓存类。

这个包的关键部分是类 `TftpConnection` 和 `TftpServerConnection`。实际的工作发生在类 `TftpReadConnection` 和 `TftpWriteConnection`。其余的类要么是用户接口（`Tftp` 和 `Tftpd`），要么是 I/O 支持类（`TftpLogStream` 和 `LineBufferedOutputStream`）。

4.1.6 TFTP 与 FTP 的对比

在第 6 章里，你将会看到 FTP 的介绍。用户经常可以使用这个服务在计算机之间传输文件，那为何还有 TFTP 呢？有一个原因，TFTP 非常简单，并且经常用于简单的无盘工作站计算机检索必要的文件。但是，TFTP 的安全性是完全不存在的。

你可以改变 TFTP 服务器，只接收来自于已知的 IP 地址的请求，这不一定很安全，但它提供了某种程度上的保护。当然，没有什么事物是十全十美的；黑客可以采取伪造的 IP 地址（一个有名的计算机骗局），但是它还是能阻止一些临时黑客的攻击。

然而，TFTP 协议容易扩展成客户化的应用程序。假设你有一台计算机采集来自一个温度传感器的信息，并将之存进一个文件里，每天晚上需要通过网络将这个文件发送到一个远程计算机那儿。用 TFTP 实现这个功能将非常简单，除此以外，你可以很容易地扩展 TFTP 协议，以提供你想实现的任何程度的安全性。

例如，改变服务器用于监听的端口号会提供一个比较适中的安全度，也可以让客户端在 WRQ 或者 RRQ 包后追加密码来保证安全。为获取最大安全度，你可以决定加密数据包的内容，使用简单的密钥技术或者更高级的加密技术。为何在 TFTP 已经存在的情况下还要发明一种新协议来传输文件呢？

在中国水利水电出版社网站上，你可以找到另一个 TFTP 类，名为 `TftpCustom`。这个类除了提供一种简单的加密方式以外，与类 `TftpSocket` 没有区别。特别地，文件的开头附近有这么一行：

```
final static byte crypt=(byte)0xA5;
```

客户端和服务端必须都遵从这个数。接着，当形成数据包时，那段代码是这样的：

```
for (int idx=0;idx<l;idx++)
    bytes[len+idx]=(byte) (fbuf[idx]^crypt);
// This line was from the old code
//System.arraycopy(fbuf,0,bytes,len,1);
```

在原始代码里边，`arraycopy` 将 `fbuf` 中的字节移到字节数组中，组成数据报的数据。加密版的程序使用一个 `for` 循环，将每个字节执行异或运算，再将它复制到字节数组中。

在接收端，程序将反过来处理：

```
for (int idx=4;idx<pack.getLength();idx++)
    byt[idx]=(byte) (byt[idx]^crypt);
try {
    fileout.write(byt,4,pack.getLength()-4);
}
catch (IOException e) { }
```

当然，这个加密方案是非常简单的。可是，它将能阻止一般的临时黑客攻击，如果你需要更高等级的安全度，可以自由地设计复杂的加密方案。

4.2 快速解决方案

4.2.1 探寻 TFTP 的规范

TFTP 协议在 RFC1350 里有定义，取代了 RFC783，可以下载这个文档的一个网址是 <http://www.faqs.org/rfcs/rfc1350.html>。

4.2.2 创建一个 TFTP 类

因为服务器端与客户端几乎相同，可以写一个类来处理这两种情形，这样比较有意义（见清单 4.5），类 TftpSocket 有方法 fsend 和 freceive 供客户端使用，以及一个入口点 tftpd，使这个类能够充当服务器。

这个类也有简化了的 main 函数，以供测试。在没有参数的情况下，这个类就是一个服务器程序。如果要让这个程序成为客户端，需要提供文件名、文件类型（netascii 或者 octet）和主机名，这样可以从服务器那里接收文件。如果再加上参数 S，程序就能发送文件到服务器中。

清单 4.5 完整的 TFTP 类

```
// TFTP Class -- Williams
import java.net.*;
import java.util.*;
import java.io.*;

public class TftpSocket extends DatagramSocket
    implements Runnable {
    final static int NONE=0; // local
    final static int RRQ=1;
    final static int WRQ=2;
    final static int DAT=3;
    final static int ACK=4;
    final static int ERR=5;
    final static int UNKNOWN=6; // local code
    final static int TIMEOUT=7; // local code
```

```

protected String fnstring;
protected byte[] fn;
protected byte[] ftype;
protected InetAddress hostaddr;
protected int port=69;
protected byte[] bytes = new byte[516];
protected byte[] bytesin = new byte[1024];
protected int block=0;
protected FileInputStream filein;
protected FileOutputStream fileout;
protected boolean EOF=false;
protected int serverCode=0; // operation requested

// Constructors (same as DatagramSocket)

public TftpSocket(String filename, String type, String host)
    throws UnknownHostException, SocketException {
    create(filename,type,host);
}

public TftpSocket(int sport, String filename, String type, String host)
    throws UnknownHostException, SocketException {
    super(sport);
    create(filename,type,host);
}

public TftpSocket(int sport, InetAddress ia,
    String filename, String type, String host)
    throws UnknownHostException, SocketException {
    super(sport,ia);
    create(filename,type,host);
}

// private constructor (used by server only)
private TftpSocket() throws SocketException { }

// this private function does the work for
// all the client constructors
private void create(String filename, String type, String host)
    throws UnknownHostException, SocketException
{
    try {
        fn=filename.getBytes("ISO-8859-1");
        ftype=type.getBytes();
    }
    catch (Exception e) {
        fn=filename.getBytes();
        ftype=type.getBytes();
    }
}

```



```

        fnstring=filen;
        setSoTimeout(1500); // 1.5 second time-out
        hostaddr=InetAddress.getByName(host);
    }

// Send a specific error packet
protected boolean sendErrPacket(int code,String msg) {
    bytes[0]=(byte)((ERR>>8)&0xFF);
    bytes[1]=(byte)(ERR & 0xFF);
    bytes[2]=(byte)((code>>8)&0xFF);
    bytes[3]=(byte)(code & 0xFF);
    System.arraycopy(msg.getBytes(),0,bytes,4,msg.length());
    bytes[4+msg.length()]=0;
    DatagramPacket pack =
        new DatagramPacket(bytes,5+msg.length(),hostaddr,port);
    try {
        send(pack);
    }
    catch (Exception e) {
        return false;
    }
    return true;
}

// Send a specific acknowledgment
protected boolean sendAck(int block) {
    bytes[0]=(byte)((ACK>>8)&0xFF);
    bytes[1]=(byte)(ACK & 0xFF);
    bytes[2]=(byte)((block>>8)&0xFF);
    bytes[3]=(byte)(block & 0xFF);
    DatagramPacket pack = new DatagramPacket(bytes,4,hostaddr,port);
    try {
        send(pack);
    }
    catch (Exception e) {
        return false;
    }
    return true;
}

// Send a packet (generic error or current block ACK)
protected boolean sendPacket(int ptype) {
    int len=2;
    bytes[0]=(byte)((ptype>>8)&0xFF);
    bytes[1]=(byte)(ptype & 0xFF);
    switch (ptype) {
        case RRQ:
        case WRQ:

```

```

        System.arraycopy(fn, 0, bytes, 2, fn.length);
        bytes[2+fn.length]=0;
        System.arraycopy(ftype, 0, bytes, 3+fn.length, ftype.length);
        bytes[3+fn.length+ftype.length]=0;
        len+=fn.length+ftype.length+2;
        break;

    case DAT:
    case ACK:
        int l;
        bytes[2]=(byte) ((block>>8)&0xFF);
        bytes[3]=(byte) (block&0xFF);
        len+=2;
        // more for file
        if (ptype==DAT) {
            byte [] fbuf=new byte[512];
            try {
                l=filein.read(fbuf, 0, 512);
            }
            catch (IOException e) {
                l=0; // assume EOF
            }
            // load data
            System.arraycopy(fbuf, 0, bytes, len, l);
            len+=l;
            if (l!=512) EOF=true;
        }
        break;

    case ERR:
        bytes[2]=bytes[3]=bytes[4]=0;
        len=5;
        break;

    default:
        return false;
    }
    DatagramPacket pack = new DatagramPacket(bytes, len, hostaddr, port);
    try {
        send(pack);
    }
    catch (Exception e) {
        return false;
    }
    return true;
}

// Process a received packet

```

```
protected int rcvPacket() throws SocketException {
    DatagramPacket pack = new DatagramPacket(bytesin,1024);
    byte [] byt;
    boolean status=false;
    try {
        receive(pack);
        status=true;
    }
    catch (IOException e) { }
    if (!status) return TIMEOUT;
    // we only process ACK, DAT, ERR
    port=pack.getPort();
    byt=pack.getData();
    int code=(byt[0]<<8)+byt[1];
    int blockin;
    switch (code) {
        case RRQ:
        case WRQ:
            return code;
        case ACK:
            // check block #
            blockin=(byt[2]<<8)+byt[3];
            if (blockin!=block) return NONE;
            block++;
            return ACK;
        case DAT:
            // check block # and store
            blockin=(byt[2]<<8)+byt[3];
            if (blockin>block) return UNKNOWN;
            if (blockin<block) {
                sendAck(blockin); // resend ACK
                return NONE;
            }
            // block number matches
            try {
                fileout.write(byt,4,pack.getLength()-4);
            }
            catch (IOException e) { }
            if (pack.getLength()<516) EOF=true;
            sendPacket(ACK); // ack current
            block++;
            return DAT;

        case ERR:
            // should do something smart with the error message
            String errm = new String(byt,4,pack.getLength());
            System.out.println(errm);
    }
}
```

```
        return ERR;
    default:
        return UNKNOWN;
    }
}

// Send a file
public boolean fsend() throws SocketException {
    boolean rv;
    try {
        filein = new FileInputStream(fnstring);
    }
    catch (Exception e) { return false; }
    try {
        int n;
        int retry=0;
        block=0;
        rv=sendPacket(WRQ);
        if (!rv) return rv;
        while (!EOF) {
            do {
                n=rcvPacket();
                if (n==TIMEOUT) {
                    if (++retry>5) return false;
                    if (block!=0)
                        if (!sendPacket(DAT)) return false;
                    continue;
                }
                if (n==ERR) return false;
                if (n==UNKNOWN) {
                    sendErrPacket(0,"Unknown error");
                    return false;
                }
            } while (n!=ACK);
            if (!sendPacket(DAT)) return false;
        }
        return true;
    }
    finally {
        streamClose(filein); // close no matter what
    }
}

// Get a file
public boolean freceive() throws SocketException {
    boolean rv;
    int n;
```

```
    try {
        fileout=new FileOutputStream(fnstring);
    }
    catch (Exception e) { return false; }
    try {
        int retry=0;
        block=1;
        rv=sendPacket(RRQ);
        if (!rv) return rv;
        while (!EOF) {
            do {
                n=rcvPacket();
                if (n==TIMEOUT) {
                    if (retry++>5) return false;
                    continue;
                }
                retry=0; // reset retry count
                if (n==ERR) return false;
                if (n==UNKNOWN) {
                    sendErrPacket(0,"Unknown error");
                    return false;
                }
            } while (n!=DAT);
        }
        return true;
    }
    finally { // close in all cases
        if (!streamClose(fileout)) return false;
    }
}

// this is the server "thread"
public void run()
{
    if (serverCode==WRQ) {
        // Set up output stream
        try {
            fileout=new FileOutputStream(fnstring);
        }
        catch (IOException e) {
            sendErrPacket(2,"Access violation");
            return;
        }
        // send ACK0
        sendAck(0);
        block=1;
        // wait for DATA
        int n;
```



```
do {
    try {
        n=rcvPacket();
    }
    catch (SocketException e) { n=ERR; }
    if (n==ERR)
        streamClose(fileout);
        return;
    }
    if (n!=DAT&& n!=NONE) {
        sendErrPacket(0, "?");
        streamClose(fileout);
        return;
    }
} while (!EOF);
}
if (serverCode==RRQ) {
    // set up input stream
    try {
        filein=new FileInputStream(fnstring);
    }
    catch (IOException e) {
        sendErrPacket(1, "File not found");
        return;
    }
    // send DATA
    block=1;
    int retry=0;
    do {
        int n;
        if (!sendPacket(DAT)) {
            sendErrPacket(0, "?");
            streamClose(filein);
            return;
        }
        do {
            try {
                n=rcvPacket();
            }
            catch (SocketException e) { n=ERR; }
            if (n==ERR) {
                streamClose(filein);
                return;
            }
            if (n==UNKNOWN||n==RRQ||n==WRQ||n==DAT) {
                sendErrPacket(4, "Illegal");
                streamClose(filein);
                return;
            }
        }
```

```

    }
    if (n==TIMEOUT && ++retry>5) {
        sendErrPacket(0, "Timeout");
        streamClose(filein);
        return;
    }
    if (n==ACK) {
        block++;
        retry=0;
    }
    } while (n!=ACK&& n!=TIMEOUT);
} while (!EOF);
}
}

// Interface for server
public static void tftpd() {
    DatagramSocket sskt;
    try {
        sskt = new DatagramSocket(69);
    }
    catch (SocketException e)
        System.out.println("Unable to start server: " + e);
    return;
}
byte [] buf = new byte[1024];
DatagramPacket pkt=new DatagramPacket(buf,1024);
while (true) {
    try {
        // wait for WRQ or RRQ -- all others are errors
        sskt.setSoTimeout(0);
        sskt.receive(pkt);
        int port=pkt.getPort();
        byte[] byt=pkt.getData();
        int code=(byt[0]<<8)-byt[1];
        if (code!=WRQ && code!=RRQ) {
            byte[] errary = new byte[4];
            errary[0]=(byte)((ERR>>8)&0xFF);
            errary[1]=(byte)(ERR & 0xFF);
            errary[2]=(byte)0;
            errary[3]=(byte)0;
            errary[4]=0;
            pkt.setData(errary);
            sskt.send(pkt);
            continue;
        }
        // Start new thread (use private constructor)
        TftpSocket worker=new TftpSocket();
    }
}

```

```

        int zlen,zlen1;
        worker.port=pkt.getPort();
        worker.hostaddr=pkt.getAddress();
        for (zlen=0;byt[zlen+2]!=0;zlen++);
        worker.fnstring=new String(byt,2,zlen);
        worker.fn=worker.fnstring.getBytes();
        for (zlen1=0;byt[zlen+zlen1+2]!=0;zlen1++);
        worker.ftype=new String(byt,zlen+2,zlen1).getBytes();
        worker.serverCode=code;
        new Thread(worker).start();
    }
    catch (IOException e) { /* ignore */ }
}

// Handy function to close a stream
private boolean streamClose(InputStream s) {
    boolean rv=true;
    try {
        s.close();
    }
    catch (Exception e) { rv=false; }
    return rv;
}

// Handy function to close a stream
private boolean streamClose(OutputStream s) {
    boolean rv=true;
    try {
        s.close();
    }
    catch (Exception e) { rv=false; }
    return rv;
}

// Test main
// provide file name, file type, host on command line
// follow with an S to send R (or nothing) to receive
public static void main(String [] args) throws Exception {
    if (args.length==0) {
        System.out.println("Starting server");
        new Tftpd().start();
        return;
    }
    TftpSocket sock=new TftpSocket(args[0],args[1],args[2]);

```

```
        if (args.length<=3||!args[3].equals("S")) {
            System.out.println("Client request "+sock.freceive());
        }
        else {
            System.out.println("Client request " + sock.fsend());
        }
    }
}

// Helper thread for test main
class Tftpd extends Thread {
    public void run() {
        TftpSocket.tftpd();
    }
}
```

4.2.3 创建一个 TFTP 的客户端应用程序

如果要将清单 4.5 中的 TFTP 代码改成客户端程序，可以将所有服务器相关代码去掉，即可以移去类 `Tftpd`，`TftpSocket.run` 方法和 `TftpSocket.tftpd` 方法。

如果你计划自己编写代码，客户端必须执行下述步骤：

1. 发送 RRQ 或者 WRQ 请求到服务器的 69 端口。
2. 等待一个数据包（如果是读）或者 ACK 包（如果是写）。这个包将包含一个 69 以外的新端口号。
3. 接收到数据包，就以 ACK 包响应，接收到 ACK 包，就发送下一个数据包。
4. 准备处理超时错误或者是 ERR 包。

4.2.4 创建一个 TFTP 的服务器端应用程序

你可以修改清单 4.5 中的代码，使之只充当服务器。只需去掉方法 `main`、`fsend` 和 `freceive`。如果想自己创建服务器，要包括下述步骤：

1. 监听 69 端口上的 RRQ 或者 WWQ 请求。
2. 对每一个合法请求，创建一个套接字（也可能是一个新线程）。对 RRQ 请求，以初始的数据包响应，对 WWQ 请求，发出对第 0 块的确认。
3. 接收到 ACK 包，以下一个数据包来响应，接收到数据包，以 ACK 包来响应。
4. 准备处理超时错误或者 ERR 包。

4.2.5 使用 GNU 的 TFTP 类

在 <http://www.gnu.org/software/java/java-software.html> 可以找到 TFTP 的实现。其中的那些类也执行客户端和服务器的 TFTP 功能。

创建客户端连接，可以这样写：

```
conn = new TftpConnection(address, rFilen, lFilen, requesttype, debug);
conn.setSocketTimeout(5000); // 5 second time-out
Thread th = new Thread(conn);
th.start();
```

当然，必须为这段代码导入包 `gnu.inet.tftp` 才能工作，`TftpConnection` 的构造函数的参数如下：

- **Address**——远程服务器的地址（一个 `InetAddress` 对象）。
- **RFilen**——远程文件名。
- **LFilen**——本地文件名。
- **Requesttype**——两种请求 `TFTP_ReadRequest` 和 `TFTP_WriteRequest` 之一。
- **Debug**——调试等级（不调试的等级为 0）。

创建一个服务器，下述代码基本上足够了：

```
Tftpd server;
server = new Tftpd();
server.processConnections();
```

4.2.6 配置 GNU 的 TFTP 服务器

GNU 服务器使用一个属性文件来指定其操作参数，详细指定如下：

- **RootDirectory**——所有文件请求的根目录。
- **Logging**——值设为 1，用于创建日志。
- **LogFile**——日志文件名。
- **LogStdout**——如果值为 1，只将日志创建到标准输出（`stdout`），而不创建到文件里。
- **SocketTimeout**——在超时之前等待套接字上输入的时间（单位为毫秒）。
- **MaxRetries**——重发一个包的次数。
- **DebugMode**——值设为 1，以调试消息。

第5章 Telnet 协议

5.1 深入介绍

我从未完全接受 WIMP (Windows 图标、鼠标、光标) 接口, 如果你真的认为使用鼠标比较直观, 可以试着教一个小孩如何使用它。他们会将它从桌子上拿起, 放到显示器上, 或者是用它胡乱操作。即使我的母亲, 现在非常喜爱计算机向导 (wizard), 刚开始也对鼠标的使用方法感到困惑不解。

这几年我原来的看法有了些转变, 使用鼠标也确实比过去多一些。当然, 现在你很难避免使用鼠标, 但是当 Internet 刚出现的时候, 情况就不同了, 在那个时候, 使用一个带有 110 波特调制解调器的电传打字机终端是很常见的。

现在看起来那时的速度实在太慢, 但是很多人利用这些条件做了很多工作。即使当玻璃的电传打字机终端 (也就是说, 终端用的是显示器而不是打印机) 出现时, 各个人的操作就有些不同了。没有明显的标准, 很难使一个供应商提供的终端与另一个公司提供的计算机进行对话, 有些终端 (特别是来自于 IBM) 甚至不使用 ASCII 字符。

现在, 终端越来越标准化了, 如果你对终端的使用经验主要是基于 PC 终端的模仿, 你可能已经遇上了 VT 类型的终端。这种类型的终端使用由数据设备公司 (DEC) 的 VT 终端所流行使用的控制码; 当然, 康柏曾经并购过 DEC 公司, 这些终端使用 escape 序列, 也就是说, 一个 escape 字符后跟随着一个或者多个命令来控制光标和屏幕显示。

这些终端最重要的特征是当你按下一个键时, 可以期待它立即将一个字符传给主机。对另一些终端则不是这样。有些惠普和 IBM 的计算机支持终端收集整个屏幕的数据, 并将它们立即传输出去。也有些终端不能处理全双工操作, 一个半双工的终端会让你通过键盘输入或者让主机访问显示器或打印机, 但这两种工作不能同时进行。

为统一这些不同类型的终端, Internet 在 RFC854 中定义了 Telnet 协议 (还有其他一些 RFC 定义了它的规范)。Telnet 允许任意的一种终端使用网络登录进任意的一个远程计算机, 很多开发人员使用 Telnet 访问远程服务器, 但是, 有很多协议使用 Telnet 协议的一些或者全部作为基础。

Telnet 要依赖于一些 Java 并不支持的套接字操作, 因此, 不可能使用 Java 完全实现 Telnet 协议。但是, 从实际应用出发, 不用 Java 不支持的那一部分 (操作), 也足够我们用了。

5.1.1 Telnet 回顾

刚一看，实现 Telnet 似乎很简单，只要在 23 端口上打开一个套接字并交换数据，对吗？如果有一个通用的终端，那就会很简单，但是，因为终端（和主机）的类型不同，事情就变得复杂了。

理论上，Telnet 并不区分终端和计算机，当然实际上很少见，但是两个终端可以通过 Telnet 互相通信，而不需要干预计算机。连接上以后，双方都伪装成为网络虚拟终端（NVT）。NVT 是一个具有任何终端特征的终端原型，尽管 Telnet 程序可能不得不将 NVT 的代码和命令翻译给本地设备。双方可以就他们使用的那些特殊特征达成协议，虽然 NVT 能力有限，但是双方可以使用任何特征，只要它们都同意使用。

5.1.2 NVT 回顾

网络虚拟终端（NVT）是对基本终端的模拟。这种终端有哪些特征呢？概念上，终端有一个键盘和打印机，它们都使用 8 位一个字节的 7 位 ASCII 码。缺省情况下，NVT 会将你键入 Telnet 连接和打印机的任何字符发送出去，那意味着 Telnet 连接的另一端就不必将那些字符回显。

缺省情况下 NVT 将缓冲字符，直到本行结束，直到 NVT 没有更多的缓冲，或者直到程序或者用户产生更多的本地定义的信号传输（例如，一个发送键）。

缺省情况下，NVT 是一种半双工设备。这也是规范中不区分一台计算机和一个终端的体现，主机在需要更多输入的情况下，应该发送一个特殊的继续（GA）信号，这允许真正的半双工终端来对它们的键盘“解锁”，这个规范同时也声明了，它并不想要终端在每一行末发送一个 GA 信号，但是，如果两个终端是在直接通信，它们将有必要在每一行后边发送 GA 信号（在缺少其他选择的情况下）。RFC 建议为用户提供一种手工发送 GA 信号的方式。

RFC 规定只有三个特殊字符 NVT 必须处理。NUL（码值为 0）、换行符（LF；码值为 10）、以及回车符（CR；码值为 13），它们都有自己常用的含义。特别地，NUL 字符不做任何事情。LF 字符将打印机头下移，并保持相同的水平位置，CR 字符会让打印机移到当前行的左边空白处。

NVT 考虑到了由 CR 和 LF 一起定义一个新行的行结束约定，如果你想发送一个真正的 CR 字符（也就是说，没有移到新行，而只是移到本行左边），你就必须发送 CR 与 NUL 字符的组合，用这种方法，必须模拟真的 CR 的系统（例如，通过 backspace 键）在碰到 CR 与 LF 组合的情况下就不必这么做。Telnet 程序应该删去到来的 NUL 字符，如果 NUL 字符后紧跟着 CR 字符，并且没有将附加的字节传递给底层程序。

另外，NVT 可以响应其他几个命令（见表 5.1）。但是，按规定对几个有趣的问题要表示“沉默”，如 tab 键该停留在终端上的哪个位置或者如何去改变它们。

表 5.1 可选的 NVT 字符

字符	码值	说明
BEL	7	产生一个信号（通常听得见的），但不移动光标
BS	8	将光标向左移动一格
HT	9	移到另一个水平 tab 键停止的位置
VT	11	移动下一个垂直 tab 键停止的位置
FF	12	形成回卷（移至下一页）；在非打印终端上常常清除屏幕

除这些特殊的字符外，Telnet 也定义了一系列特殊的 Telnet 命令，它们都是一些字节序列，并以解释为命令（IAC）字节开始。这个字节值为 255。如果你真的需要向数据流中发送 255，简单地将它发送两次即可。否则，下一个字节（可能是一系列字节）将组成一个特殊的 Telnet 命令。这里有很多命令是对一些互相同意的选项进行协商。其他的一些命令允许中断输出，并执行其他特殊的功能（它们与终端上的数据显示并没有具体关联）。

5.1.3 特殊命令

Telnet 协议支持 16 种特殊命令，在表 5.2 中可以看到。其中 DO、DONT、WILL、SB、SE 和 WONT 命令与协商有关，后边将可以看到它们是如何工作的。

表 5.2 Telnet 命令

命令	码值	说明
IAC	255	解释为命令（在数据里发送的是 255 连着发送两次）
DONT	254	请求接收者停止执行一个选项
DO	253	请求接收者开始执行一个选项
WONT	252	拒绝执行一个选项
WILL	251	同意执行一个选项
SB	250	子选项协商
GA	249	继续（半双工）
EL	248	删除行
EC	247	删除字符
AYT	246	你在那儿吗
AO	245	放弃输出
IP	244	中断处理
BRK	243	阻塞
DM	242	数据标志
NOP	241	无操作
SE	240	子项协商结束

GA 命令，像先前提到的一样，通知接收者它可以传输了。你将会看到，在现在的使用中，发送方和接收方将协商好，这样哪一边都不用发送 GA 命令。我想你也不会经常看到 NOP（无操作）命令。

EC 和 EL 命令是要删除最后一个字符或者是最后一行。并不是所有的终端都能有效地执行该命令，因此你不能确信它们会产生什么结果。例如，一些老的打印终端可能通过在当前行末打印一个反斜线符号并启动一个新行来表示 EL 命令。

AYT 命令表示“你在那儿吗”。这只是一个简单的查询，以确认对方还在接收。接收方应该发出一些打印的内容作为响应。

余下的命令通常来源于终端。BRK、IP 和 AO 允许你试图测试和控制运行在远程主机上的程序。主机应该对这些命令作出尽可能快的反应。

这些快速响应命令给 Java 造成了一点问题。RFC 中规定当你发送一个快速响应命令时，应该使用紧急 TCP 包（有时称为带外数据）发送一个 DM 命令。不幸的是，Java 至少在 JDK1.3 版本中没有方法来处理这些紧急数据包。

当错误处理不相符并且终端已经将 Telnet 程序缓存的数据发送出去的时候，就会出现为题。当 Telnet 程序看到紧急的 DM 命令，它应该检查缓存中是否会有 AYT、IP 和 AO 命令，这也会引起缓存字符的丢失，造成负面影响。

Java 程序不能发送或者接收带外数据，因此当它们进入数据流的时候，基于 Java 的 Telnet 程序只能处理这些特殊的字符。如果数据被缓存，优先级命令将只能等待，直到它们出现在数据流中。

不要忘了所有的 Telnet 命令必须跟在 IAC 前缀（255）后边，如果你想在数据流中发送 255 这个值，可以连续发送两个 IAC 字符。

5.1.4 要协商的地方

Telnet 连接的两方都可以请求改变缺省的 NVT 参数。但是，Telnet 规范不允许两方强制执行任何选项，简单地处理 NVT 模拟就可以满足 RFC 中的要求了。

看看字符回显的例子，很多串行终端期望主机将字符回显给终端，这样会给用户一个肯定的反馈，即到达的字符是正确的。但是，如果网络连接较慢或者终端缓存整行数据时，回显字符就没有什么意义了。反之，如果网络连接较快，并且终端是面向字符的，就可能需要进行回显处理了。

RFC857 定义回显处理的选项对应的码值为 1，要请求连接的另一端回显，可以发送特殊命令 DO 1。另一端可以发送 WILL 1 命令表示它愿意回显。

意识到两方都可以发出加显过程请求是很重要的。换句话说，在终端发送 DO 命令之前，主机可以通过发送 WILL 1 来“广播”它愿意回显，顺序是重要的。如果主机不愿意回显，它会发送 WONT 命令，终端也可以通过发送 DONT 1 命令表示它不需要字符回显。

记住 Telnet 规范并不区分主机和终端。因此，主机请求终端回显同样合法。通常，主机

不需要回显，因此在连接之后，你可能看到下述内容：

```
HOST: WILL ECHO
TERMINAL: DO ECHO
TERMINAL: WILL ECHO
HOST: DONT ECHO
```

前边的序列意指主机可以回显字符给终端，但终端不能回显字符给终端，记住开始由终端请求回显同样合法：

```
TERMINAL: DO ECHO
HOST: WILL ECHO
```

在第一个例子里，你可以考虑 DO 命令是对 WILL 命令的通知（或者响应）。在第二个例子里，DO 是命令，而 WILL 则充当响应。

5.1.4.1 循环的防止

因为 WILL 和 DO 几乎是可互换的，必须谨防无穷循环。例如，假设主机发送一个 WILL，终端发送一个 DO 来响应它，主机就不能用另一个 WILL（它会导致客户端发送另一个 DO 命令来响应，形成无穷循环）来响应客户端的 DO。

要防止这种循环，应该遵守下述规则：

- 只能请求可选状态的改变，不能只是简单地声明自己当前的状态。
- 如果接收到一个请求，要求你“改变”到你当前所在的状态（即没有发生状态改变），无须响应这个请求。换言之，如果你正在回显字符，同时接收到 DO ECHO 命令，就忽略它。

自然，任何改变状态的请求只影响这个命令之后的字符传输。很可能除了在连接开始时以外，不想改变回显的状态，但是，在整个过程的中间，可能改变其中某些状态有一定意义。

像其他 Telnet 命令一样，上述这些命令需要 IAC 前缀。例如，一个 WILL ECHO 命令，实际上是 255,251,1（WILL 是命令 251）。

5.1.4.2 其他选项

还有很多 RFC 文档定义了 Telnet 服务器和客户端互相一致的各种可能选项（见表 5.3）。如果终端（或者主机）一次需要处理一行数据，可以试用协商选项 34（由 RFC1184 定义）。它取代了旧的行模式。旧的行模式是指，回显状态为 on 与禁止继续选项为 on 两者之一出现时，Telnet 将每次发送单行数据。

表 5.3 常见 Telnet 选项

选项序号	RFC	说明
1	857	回显字符
3	858	禁止 GA（继续）
5	859	状态
6	860	分时标志

续表

选项序号	RFC	说明
24	1091	终端类型
31	1073	窗口大小
32	1079	终端速度
33	1372	远程流控制
34	1184	行模式
36	1408	环境变量

提示: 可以经常使用 Telnet 客户端显示你发送和接收的命令。绝大多数 Unix 的 Telnet 程序(以及 Windows 的 Cygwin Telnet 程序)会显示客户端和主机之间的交互过程。具体工作如何不同依赖于你的 Telnet 程序。

下面是 Cygwin Telnet 程序在发出某连接选项命令之后的一些样例输出:

```
$ telnet
telnet> toggle option
Will show option processing.
telnet> open guardian
Trying 192.244.69.30...
Connected to guardian
Escape character is '^]'.
SENT DO SUPPRESS GO AHEAD
SENT WILL TERMINAL TYPE
SENT WILL NAWS
SENT WILL TSPEED
SENT WILL LFLOW
SENT WILL LINEMODE
SENT WILL NEW-ENVIRON
SENT DO STATUS
RCVD DO OLD-ENVIRON
SENT WONT OLD-ENVIRON
RCVD DO TERMINAL TYPE
RCVD WILL SUPPRESS GO AHEAD
RCVD DO NAWS
SENT IAC SB NAWS 0 80 (80) 0 25 (25)
RCVD DO TSPEED
RCVD DO LFLOW
RCVD DONT LINEMODE
RCVD DONT NEW-ENVIRON
RCVD WONT STATUS
RCVD IAC SB TERMINAL-TYPE SEND
SENT IAC SB TERMINAL-TYPE IS "xterm"
RCVD IAC SB TERMINAL-SPEED SEND
SENT IAC SB TERMINAL-SPEED IS 38400,38400
```

```
RCVD WILL ECHO
SENT DO ECHO
RCVD DO ECHO
SENT WONT ECHO
RCVD DONT ECHO
login:
```

5.1.4.3 子选项

某些选项在使用时需要附加的信息。首先，连接双方要协商这个选项，在达成一致之后，可以交换 **SB** 命令（子选项）。例如，终端可能需要告知主机它的终端类型，这样会允许主机发送终端相关命令序列来更改显示。

首先，客户端会告知自己要提供信息的意愿（使用选项 24）：

```
IAC WILL 24
```

主机接着响应并发出 **DO** 命令来接受这个选项：

```
IAC DO 24
```

接着，主机会使用 **SB** 命令请求子选项的值，请求中的 1 表明主机想要知道子选项的值。

在任何 **SB** 命令之后，会看到 **SE** 命令：

```
IAC SB 24 1
```

```
IAC SE
```

终端会以下述内容响应（带引号的数据是串文字）：

```
IAC SB 24 0 "VT52"
```

```
IAC SE
```

SB 命令里的 0 表明终端在提供值（这里是 VT52）。当然，不同的子命令会有不同的数据。

5.1.5 Telnet 实践

Telnet 的 RFC 文档为 Telnet 描绘出总的框架图，而最新的 Telnet 程序提供的功能远多于那些基本功能。特别地，很少能见到实用的 Telnet 客户端程序不模拟一些流行的终端（甚至很多种不同类型的终端）。它们如此流行，以至于很多主机假设你至少有能力显示特殊字符或者移动光标。

有时你连接到的程序也会影响你接收数据的方式。例如，即使 Telnet 不向你的终端回显字符，你连接到的程序也会回显。

典型的一个例子，可以把一个 Telnet 客户端连接到的程序看作是一个 Unix 的命令操作外壳（shell）。但在某些情况下 Telnet 连接比较有用，例如，一个数据库服务器可能通过 Telnet 接受查询。很多会议系统和 MUD（多用户城堡,Multi-User Dungeon）游戏可以使用 Telnet。

随着对安全问题的逐渐关注，很多站点正竭力逐步停用 Telnet。缺省情况下，Telnet 在网络上并不隐藏你的任何数据（包括密码）。这意味着任何能够截取你的网络通信的人都可能危及你的网络安全。可以使用几种方法对套接字数据进行加密（例如，安全套接字即 SSL），并且这些都可以在 Telnet 上工作。但是，实际上，绝大多数站点使用 SSH（安全 Shell），使安全登录更便利。

SSH 使用特殊的加密技术来加密服务器与客户端之间的所有数据。也能向双方鉴别并证明某台机器并不是简单的伪装成你认为你正使用的计算机。

5.1.6 一个基本的 Java 客户端

一个合适的 Telnet 客户端程序应该准备随时发送和接收数据，因此使用线程会好一些。我想创建一个用途广泛的类，处理所有的协议关系，这样可以从新类继承，它只包含你需要的专门处理，其结果是类 TelnetTTY，如清单 5.1 所示。

清单 5.1 从基类创建 Telnet 服务器端应用

```
/* Telnet TTY class -- Williams
   This class provides a base for writing objects
   that do the telnet protocol.

   Bare minimum options. Assumes host will accept
   "go ahead suppression"

*/
import java.io.*;
import java.net.*;

public class TelnetTTY extends Thread {
    // Telnet constants
    final static int TAC=255;
    final static int DONT=254;
    final static int DO=253;
    final static int WONT=252;
    final static int WILL=251;
    final static int NOP=241;
    final static int OP_ECHO=1;
    final static int OP_SGA=3;

    // default NVT has echo off, go ahead on in both directions
    // however, we assume Suppress GA will work

    protected Socket sock;
    protected InputStreamReader rdr;
    protected OutputStreamWriter wrt;
    protected boolean errflag=false;
    protected IOException ioerror; // if an error, what was it?
    private boolean crflag=false;
    private boolean echoflag=false;
    protected boolean stopthread=false;
```

```

// Constructor
public TelnetTTY(String host, int port) throws IOException,
    UnknownHostException {
    TelnetTTYCons(host,port);
}

// Constructor
public TelnetTTY(String host) throws IOException,
    UnknownHostException {
    TelnetTTYCons(host,23);
}

// Core code for constructors
public void TelnetTTYCons(String host, int port) throws IOException,
    UnknownHostException {
    sock=new Socket(host,port);
    rdr=new InputStreamReader(sock.getInputStream());
    wrt=new OutputStreamWriter(sock.getOutputStream());
    start();
// some hosts won't do anything until you start talking
    sendc(IAC); sendc(DO); sendc(OP_SGA);
    sendc(IAC); sendc(WILL); sendc(OP_SGA);
}

// read error flag and information
public boolean getErrorFlag() {
    return errflag != (ioerror!=null);
}

public IOException getErrorException() {
    return ioerror;
}

// send a character to the remote
public void send(int c) throws IOException {
    if (c=='\n') wrt.write('\r'); // LF=>CR LF
    wrt.write(c);
    if (c=='\r') wrt.write(0); // CR=>CR NUL
    if (c==IAC) wrt.write(IAC); // double IACs in stream
    wrt.flush();
}

// Send an entire string
public void send(String s) throws IOException {
    for (int i=0;i<s.length();i++) {
        int c=(int)s.charAt(i);
        if (c=='\n') wrt.write('\r'); // LF=>CR LF
        wrt.write(c);
        if (c=='\r') wrt.write(0); // CR=>CR NUL
    }
}

```

```
        if (c==IAC) wrt.write(IAC);
    }
    wrt.flush();
}

// This thread listens for incoming characters
public void run() {
    int c;
    while (!stopthread) {
        c=getChar();
        if (c==-1||ioerror!=null) break;
        if (c==IAC) {
            processCmd();
            if (ioerror!=null) return;
        }
        else
            processChar(c);
    }
    errflag=true;
}

// Get a character from the stream
protected int getChar() {
    try {
        int c;
        if (crflag) {
            crflag=false;
            c=rdr.read();
            if (echoflag) wrt.write(c);
            if (c!=0) return c; // ignore CR NUL
        }
        c=rdr.read();
        if (echoflag) wrt.write(c);
        if (c=='\r') crflag=true;
        return c;
    }
    catch (IOException e) {
        ioerror=e;
        return -1;
    }
}

// Send a "bare" character
private void sendc(int c) {
    try {
        wrt.write(c);
        wrt.flush();
    }
}
```



```
        catch (IOException e) {
            ioerror=e;
        }
    }

    // Process a command
    protected synchronized void processCmd() {
        int c;
        c=getChar();
        if (ioerror!=null) return;
        switch (c) {
            case IAC:
                processChar(IAC); // escaped 8-bit character
            case NOP: // nop
                return;
            case DO: // DO
                int state=WONT; // wont
                c=getChar();
                if (ioerror!=null) return;
                if (c==OP_SGA) {
                    state=WILL; // will
                }
                if (c==OP_ECHO && !echoflag) {
                    state=WILL; // will
                    echoflag=true;
                }
                sendc(IAC);
                sendc(state);
                sendc(c);
                return;

            case WILL: // WILL
                c=getChar();
                return;

            case WONT: // WONT
                c=getChar();
                if (c==OP_SGA) {
                    // hmm... we have to have SGA
                }
                return;

            case DONT:
                c=getChar();
                if (c==OP_ECHO && echoflag) {
                    echoflag=false;
                }
                sendc(IAC);
                sendc(WONT);
            }
        }
    }
}
```

```

        sendc(OP_ECHO);
    }
    if (c==OP_SGA) {
        // hmm... we have to have SGA
    }
    return;

// do more IAC commands
default: // just for debugging purposes
    System.out.println("\n[CMD: " + c + "]\n");
}
}

// process incoming characters - subclass will override
protected synchronized void processChar(int c) {
    if (c<32 && c != 10 && c !=13 && c !=9 && c !=8)
        return;
    System.out.print((char)c);
}

// Test main
public static void main(String args[]) throws Exception {
    int c;
    TelnetTTY tty = new TelnetTTY(args[0]);
    while ((c=System.in.read())!=-1) {
        if (tty.getErrorFlag()) break;
        tty.send(c);
    }
    System.out.println("Finished");
}
}

```

这个对象有两个构造函数。第一个构造函数需要主机名，并假设你使用端口 23；第二个构造函数允许指定一个端口号。因为实际上工作是一样的，一个私有函数里包含有启动客户端处理的真实逻辑。

Telnet 客户端程序可能会在几个地方不同时地发现错误或者是不完整的套接字。如果有错误发生，方法 `getErrorFlag` 返回真值。如果错误是由异常引起的，方法 `getErrorException` 会返回 `IOException` 对象。

当应用程序想发送数据到远程连接的终点时，可以使用 `send` 方法来发送单个字符或者是一个串。该方法处理任意的双 IAC 字符，这样它们不会被误解成命令。

`Run` 方法用于接收线程。线程等待字符，接着调用方法 `processChar` 或者 `processCmd`。绝大多数子类会重载方法 `processChar`，但是很少子类重载方法 `processCmd`。例如 `processChar` 简单地打印字符到系统的控制台，但子类可能会选择将字符放入图形用户接口面板的文本域里。

ProcessCmd 方法处理 WILL、WONT、DO、DONT 逻辑，它假设远程主机允许你不用 GA 字符。我还没有发现有服务器不是这样工作的。

余下的一些函数都是些基本的支持函数，也有一个测试用的 main 函数，它会连向你在命令行中指定的主机那里。甚至这个简单的 main 函数也比基本的 Telnet 客户端程序工作得相对好一些。它从系统控制台中读取字符，允许缺省的 processChar 方法打印到来的字符。

如果使用这个简单的客户端程序登录到一个 shell 上，你可能发现双字符。那是因为程序别无办法，只能显示你键入的字符，没有一个便利的方法使用系统控制台不回显你键入的字符。

5.1.7 创建一个 Telnet 服务器端应用

Telnet 服务器与客户端区别很小，从协议的角度来看，没有什么不同。当然，服务器端管理它的套接字会有所不同，因为它要监听到来的连接。当然，一个服务器端应用也将需要提供某些服务，如数据库访问。

我决定修改现存的 TelnetTTY 对象，以便它能创建服务器（见清单 5.2）。这涉及到几个新函数，具体如下：

- 一个新的构造函数会接受一个 Socket 对象。
- TelnetTTYServer 方法创建一个 ServerSocket 对象，并接受连接请求。每个连接将该对象实例化，并调用它的 connected 方法。
- 一个 connected 方法，会被子类重载。通常在这个方法里服务器会提示登录或者显示一个正式连接消息。

方法 TelnetTTYServer 不能直接创建 TelnetTTY 对象，因为你通常需要它创建一个子类。因此，这个方法需要 Class 对象来指定哪个类要实例化。程序使用了 getConstructor 方法来查找与你的对象相对应的 Constructor 对象。接着程序调用 newInstance 来创建你的对象。这与第 2 章中的 MTSERVERBase 对象使用了相同的技术。

清单 5.2 对 TelnetTTY 对象的部分修改，允许创建 Telnet 服务器

```
/* Telnet TTY class -- Williams
   This class provides a base for writing objects
   that do the telnet protocol.

   Bare minimum options. Assumes host will accept
   "go ahead suppression"

*/
import java.io.*;
import java.net.*;

public class TelnetTTY extends Thread {
    // Telnet constants
```

```
final static int IAC=255;
final static int DONT=254;
final static int DO=253;
final static int WONT=252;
final static int WILL=251;
final static int NOP=241;
final static int OP_ECHO=1;
final static int OP_SGA=3;

// default NVT has echo off, go ahead on in both directions
// however, we assume Suppress GA will work

protected Socket sock;
protected InputStreamReader rdr;
protected OutputStreamWriter wrt;
protected boolean errflag=false;
protected IOException ioerror; // if an error, what was it?
private boolean crflag=false;
private boolean echoflag=false;
protected boolean stopthread=false;
// Constructor
public TelnetTTY(String host, int port) throws IOException,
    UnknownHostException {
    TelnetTTYCons(host,port);
}
// Constructor
public TelnetTTY(String host) throws IOException,
    UnknownHostException {
    TelnetTTYCons(host,23);
}

// Core code for constructors
public void TelnetTTYCons(String host, int port) throws IOException,
    UnknownHostException {
    sock=new Socket(host,port);
    rdr=new InputStreamReader(sock.getInputStream());
    wrt=new OutputStreamWriter(sock.getOutputStream());
    start();
// some hosts won't do anything until you start talking
    sendc(IAC); sendc(DO); sendc(OP_SGA);
    sendc(IAC); sendc(WILL); sendc(OP_SGA);
}

// Server constructor
public TelnetTTY(Socket skt) throws IOException {
    sock=skt;
```

```

        rdr=new InputStreamReader(sock.getInputStream());
        wrt=new OutputStreamWriter(sock.getOutputStream());
        start();
// some hosts won't do anything until you start talking
        sendc(IAC); sendc(DO); sendc(OP_SGA);
        sendc(IAC); sendc(WILL); sendc(OP_SGA);
    }

// Static call to start server
public static void TelnetTTYServer(Class c) throws IOException {
    TelnetTTYServer(c,23);
}

// Static call to start server on any port
public static void TelnetTTYServer(Class c,int port)
throws IOException {
    ServerSocket ssock=new ServerSocket(port);
    java.lang.reflect.Constructor cons;
    Class [] arg = new Class[1];
    arg[0]=Socket.class;
// find constructor for user's class
    try {
        cons=c.getConstructor(arg);
    }
    catch (NoSuchMethodException e) {
        System.err.println("Can't find constructor");
        return;
    }
    while (true) {
        Socket [] args = new Socket[1];
        TelnetTTY svr;
        args[0]=ssock.accept();
        // create user's class
        try {
            svr = (TelnetTTY)cons.newInstance((Object [])args);
        }
        catch (Exception e) { continue; }
        svr.connected();
    }
}

// called when a client connects -- subclass will override
protected void connected() {
}

// read error flag and information
public boolean getErrorFlag() {

```



```

    return errflag != (ioerror!=null);
}

public IOException getErrorException() {
    return ioerror;
}

// send a character to the remote
public void send(int c) throws IOException {
    if (c--=='\n') wrt.write('\r'); // LF=>CR LF
    wrt.write(c);
    if (c=='\r') wrt.write(0); // CR=>CR NUL
    if (c==IAC) wrt.write(IAC); // double IACs in stream
    wrt.flush();
}

// Send an entire string
public void send(String s) throws IOException {
    for (int i=0;i<s.length();i++) {
        int c=(int)s.charAt(i);
        if (c=='\n') wrt.write('\r'); // LF=>CR LF
        wrt.write(c);
        if (c=='\r') wrt.write(0); // CR=>CR NUL
        if (c==IAC) wrt.write(IAC);
    }
    wrt.flush();
}

// This thread listens for incoming characters
public void run() {
    int c;
    while (!stopthread) {
        c=getChar();
        if (c==-1||ioerror!=null) break;
        if (c==IAC) {
            processCmd();
            if (ioerror!=null) return;
        }
        else
            processChar(c);
    }
    errflag=true;
}

// Get a character from the stream
protected int getChar() {
    try {
        int c;

```

```

        if (crflag) {
            crflag=false;
            c=rdr.read();
            if (echoflag) wrt.write(c);
            if (c!=0) return c; // ignore CR NUL
        }
        c=rdr.read();
        if (echoflag) wrt.write(c);
        if (c=='\r') crflag=true;
        return c;
    }
    catch (IOException e) {
        ioerror=e;
        return -1;
    }
}

// Send a "bare" character
private void sendc(int c) {
    try {
        wrt.write(c);
        wrt.flush();
    }
    catch (IOException e) {
        ioerror=e;
    }
}

// Process a command
protected synchronized void processCmd() {
    int c;
    c=getChar();
    if (ioerror!=null) return;
    switch (c) {
        case IAC:
            processChar(IAC); // escaped 8-bit character
        case NOP: // nop
            return;
        case DO: // DO
            int state=WONT; // wont
            c=getChar();
            if (ioerror!=null) return;
            if (c==OP_SGA) {
                state=WILL; // will
            }
            if (c==OP_ECHO && !echoflag) {
                state=WILL; // will
                echoflag=true;
            }

```

```

    }
    sendc(IAC);
    sendc(state);
    sendc(c);
    return;

case WILL: // WILL
    c=getChar();
    return;

case WONT: // WONT
    c=getChar();
    if (c!=OP_SGA) {
// hmm... we have to have SGA
    }
    return;

case DONT:
    c=getChar();
    if (c==OP_ECHO && echoflag) {
        echoflag=false;
        sendc(IAC);
        sendc(WONT);
        sendc(OP_ECHO);
    }
    if (c==OP_SGA) {
// hmm... we have to have SGA
    }
    return;

// do more IAC commands
default: // just for debugging purposes
    System.out.println("\n[CMD: " + c + "]\n");
    }
}

// process incoming characters - subclass will override
protected synchronized void processChar(int c) {
    if (c<32 && c != 10 && c !=13 && c !=9 && c !=8)
        return;
    System.out.print((char)c);
}

// Test main
public static void main(String args[]) throws Exception {
    int c;
    if (args.length==0) {

```

```

        TelnetTTYServer(TelnetTTY.class);
    }
    TelnetTTY tty = new TelnetTTY(args[0]);
    while ((c=System.in.read())!=-1) {
        if (tty.getErrorFlag()) break;
        tty.send(c);
    }
    System.out.println("Finished");
}
}

```

5.1.8 定制服务器端

缺省情况下，服务器不会做任何有趣的事情，但它会接受连接。为让它执行一些有用功能，不得不创建一个子类，如清单 5.3 所示。

清单 5.3 这个简化的 Telnet 服务器能提供状态消息以及当前时间，同时可以执行 help 和 quit 命令

```

import java.net.*;
import java.io.*;

public class TelnetServe extends TelnetTTY {
    StringBuffer buf = new StringBuffer();
    // This constructor just calls the base class
    public TelnetServe(Socket s) throws IOException {
        super(s);
    }
    // on connection, print a sign on message
    protected void connected() {
        try {
            send("Welcome to the Java Telnet Server\n");
        }
        catch (IOException e) {}
    }
    // process an incoming line
    protected void processLine(String line) {
        try {
            // this is just a simple example
            if (line.equals("status"))
                send("\nStatus OK\n");
            else if (line.equals("time"))
                send("\n"+new java.util.Date().toString()+"\n");
            else if (line.equals("help"))
                send("\nCommands are: status, time, help, quit\n");
            else if (line.equals("quit")) {
                send("\nGoodbye\n");
                sock.close();
            }
        }
    }
}

```

```

        stopthread=true;
    }
    else
        send("Unknown command\n");
    }
    catch (Exception e) {}
}

// This overrides the base class and builds a line
// that will be handled by processLine
protected synchronized void processChar(int c) {
    if (c=='\r') {
        String line=buf.toString();
        buf=new StringBuffer();
        // process line
        processLine(line);
    }
    else {
        if (c!='\n') buf.append((char)c);
    }
}

// Test main
public static void main(String args[]) throws Exception {
    TelnetTTYServer(TelnetServe.class);
}
}

```

这个类提供了重载函数 `processChar` 和 `connected`。方法 `processChar` 在 `buf` (一个 `StringBuffer` 变量) 里创建了一行。一旦完整的行准备好, 这个方法调用 `processLine` 执行实际的工作。方法 `connected` 只打印连接成功的消息。

对本例而言, `processLine` 只响应一小部分命令, `time` 命令是唯一一个执行重要处理的命令。

5.1.9 Telnet 开放源码

网上有很多 Java 的 Telnet 实现方法。可能最完整的一个是 Telnet 小程序 (在 <http://www.mud.de/se/jta/>), 不要让名字欺骗了你, 这是一个很有用的 Java 程序片段, 并且是可重用的。它支持 `plug-in` 式体系结构, 还有处理 SSH 的代码, 虽然作者 (Matthias L. Jugel 和 Marcus MeiBner) 使用它创建了小程序 (applet), 你仍有理由使用这些基本代码作为任何 Telnet 应用的基础。

包 `de.mud.telnet` 包含一个简单的类——`TelnetWrapper`, 可以使用它来编写 Telnet 程序, 而不用探测 Telnet 类到底有多深奥。清单 5.4 显示如何简单地使用它。

清单 5.4 类 `TelnetWrapper` (从源码包 `de.mud.telnet` 得到) 使得通过 Telnet 可以简单的与主机交互

```
import de.mud.telnet.*;
```



```

public class JtaTest {
    public static void main(String args[]) {
        TelnetWrapper telnet = new TelnetWrapper();
        try {
            telnet.connect(args[0], 23);
            telnet.login("alw", "xxx");
            telnet.setPrompt("/users/alw Ñ>");
            telnet.waitFor("TERM -");
            telnet.send("dumb");
            System.out.println(telnet.send("ls -l"));
        } catch (java.io.IOException e) {
            e.printStackTrace();
        }
    }
}

```

事件的顺序是简单明了的:

- **connect**——使对象与带有名字的主机在一个指定的端口上相连。
- **login**——是一个登录调用, 让一个对象在给定用户名和密码的情况下登录。
- **setPrompt**——这个调用告诉主机在提示输入的情况下, 该发送哪个串。
- **waitFor**——使用这个调用, 使得对象暂停, 直到主机发送指定的串。
- **send**——它发送你提供的串, 并返回接收的数据 (到有提示为止, 包括提示)。

可以在表 5.4 中看到那些最有用的方法。

表 5.4 用于 TelnetWrapper 的有用方法

方法	说明
connect	连向一个远程主机
disconnect	与远程计算机断连
getTerminalType	获得当前的终端类型, 即 Telnet 的 TTYPE 选项值
getWindowSize	获得终端的当前窗口大小
login	登录到远程主机
send	发送命令到远程主机; 返回发送的文本直到提示文本 (如果提示文本已经设置) (见 setPrompt)
setLocalEcho	设置本地 echo 选项
setPrompt	设置发调用 send 方法时的对象等待的提示串
waitFor	等待来自远程主机的一个串, 并返回直到 (并包括) 这个串为止的所有字符

5.2 快速解决方案

5.2.1 探寻 Telnet 协议规范

Telnet 规范在很多不同的 RFC 文档里都有所描述, RFC854 (也称为 STD0008) 定义了 Telnet 协议的内核以及 Telnet 程序必须仿效的 NVT 的特征。但实际上, 绝大多数 Telnet 程序会协商一个或多个选项, 每个选项由一个不同的 RFC 描述。RFC855 (也是 STD0008 的一部分) 定义了常用的选项, 而具体的 RFC 则包括那些特殊的选项。

最常见的选项包括如下:

- 二进制 (binary) ——这种模式允许协商二进制传输, 虽然它看起来好像 Telnet 总是使用 8 位, 记住如果没有二进制选项, 一些 Telnet 的实现可能将接收到的 8 位 NVT 字符译成另一个字符集 (如 EBCDIC)。在二进制模式下, 这种转换不会发生。但是, 使用 255 时, 必须用双字节, Telnet 命令还在起作用 (RFC0856)。
- 回显 (Echo) ——使远程主机回显字符 (RFC0857/STD0028)。
- 禁止继续——设置一个全双工连接 (RFC0858/STD0029)。
- 状态——一个 Telnet 程序可以用这一项来请求另一端的可用选项 (RFC0859/STD0030)。
- 分时标志——当一个 Telnet 程序发送另一个 DO TM 命令时, 另一方在处理 DO 命令前的任何数据之后会作出响应 (通常用 WILL TM)。即使远程主机发送 DONT TM, 它的作用还是一样 (RFC0860/STD0031)。
- 扩展选项——这一项实际上引入了 255 个更多的选项作为将来的扩充 (RFC0861/STD0032)。
- 窗口大小——即所谓的 NAWS (协商窗口大小), 这一项会让双方在 Telnet 窗口的宽和高以及字符方面上达成一致 (RFC1073)。
- 终端速度——Telnet 程序使用这一项可以设置有效的波特率, 主机可以用来控制每次发送数据的多少 (RFC1079)。
- 终端类型——Telnet 客户端可以用这一项来声明它模拟的终端类型, 主机可以用这一项选择特殊的字符来影响文本外观、设置光标位置以及执行其他特殊的操作 (RFC1091)。
- 行模式——使用这一项可让计算机之间每次都传输一行数据 (RFC1184)。
- 远程流控制——如果终端需要控制来自远程主机的输出流控制, 就可以使用这一项 (RFC1372)。
- 环境——这一项允许 Telnet 会话来设置远程主机上的环境变量 (RFC1572)。

另外, RFC652-658 定义了一些选项, 允许发送方和接收方对一些特殊字符的设置达成一致 (例如, 换行符、回车符、制表符以及其他类似字符)。

5.2.2 发送 Telnet 命令同时发送数据

每个 Telnet 命令以 IAC (255) 开始, 这使得通过 Telnet 发送数据很简单, 因为只有值为 255 的字节需要特殊处理, 如果你想发送和接收值为 255 的字节, 只需连发两次。

注意: 要确认连发两次 255 的任何字节数据不是一个 IAC。例如, 如果正协商子选项, 一个数据值为 255, 于是用两个字节, 即使你协商了二进制模式, 还是要将任何值为 255 的数据表示为双字节 (除非你指的是发送命令 IAC)。

每个 Telnet 命令 (见表 5.2) 需要不同的字节数, 很多命令需要子选项作进一步的处理。

5.2.3 模拟 NVT

Telnet 事务的两边都在模拟一个 NVT, 这个终端应该缓存尽可能多的字符 (尽可能直到行末)。它不要求远程的那一方执行字符回显, 字符集是以 8 位字节表示的 7 位 ASCII 码。

NVT 只认识一些特殊的字符, NUL (0), LF (10) 和 CR (13)。为帮助终端不用区分 CR 和行末, 实际上行结束符的发送要求发送一个 CR 符和一个 LF 符, 任何不表示行结束的 CR 符就发送成 CR NUL 的形式。

在现实应用中, 绝大多数 Telnet 程序比 NVT 功能更好。通过使用选项协商, 两边的 Telnet 程序可以就使用哪些特殊的特征达成一致。

5.2.4 协商 Telnet 的选项

如果 Telnet 程序希望使用 NVT 里没有的特征 (或者禁用 NVT 特征), 必须与远程 Telnet 程序协商, 没有一个 Telnet 程序会负责支持超出 NVT 的任何特征。

两种情况可能发生:

1. 可以要求远程系统启动 (或停止) 执行某些操作, 在这里, 发送一个 DO (或者 DONT) 命令。
2. 可以要求本地执行某些操作 (或者反过来, 要求停止执行某一操作), 在这里, 发送一个 WILL (或者 WONT) 命令。

无论是哪种情况, 必须允许另一方同意或者否定你的请求。如果发送一个 DO 命令, 对方可以发送 WILL 命令, 表明它会遵守你的请求, 或者发送 WONT 表明它不愿遵守这个要求。如果你已经发送了 WILL 命令, 远程机器会以 DO 或者 DONT 命令来响应 (当然, 这要依赖于你是否执行了对方指示的操作而定)。

5.2.5 防止循环

DO/DONT 和 WILL/WONT 命令的对称性导致了可能出现的问题。如果一个 Telnet 程序发出了 DO 命令, 另一方可能会以 WILL 响应, 但是, 源 Telnet 程序可能错误地以另一个 DO

作进一步响应，这会引发下一轮的 WILL，并导致死循环。

避免协商循环，要遵守下述原则：

- 只能请求可选状态的改变，不能只是简单地声明自己当前的状态。
- 如果接收到一个请求，要求你“改变”到你当前所在的状态（即没有发生状态改变），无须响应这个请求。换言之，如果你正在回显字符，同时接收到 DO ECHO 命令，就忽略它。

5.2.6 处理子选项

很多选项需要数据。例如，如果 Telnet 客户端需要指定它模拟的终端的类型，从某种意义上来说，它必须发送终端的 ID（例如，TTY 或者 VT52）。

看看终端类型选项（选项 24），其交互可能看起来像这样：

IAC WILL 24	本方将提供终端信息（从终端到主机）
IAC DO 24	本方将接受终端信息（从主机到终端）
IAC SB 24 1	发送给本方终端类型的子选项（从主机到终端）
IAC SE	SB 命令结束（从主机到终端）
IAC SB 24 0 “VT52”	本方类型为 VT52（从终端到主机）
IAC SE	SB 命令结束（从终端到主机）

如果发送者请求值，则 SB 命令里包含 1；如果提供值，则 SB 里包含 0。带引号的 VT52 是组成串的简单的 ASCII 字节序列。

提示：如果 SB 命令里包含值为 255 的字节，确信将此字节发送两次，以避免与 IAC 命令混淆。

5.2.7 从基类创建一个 Telnet 客户端

清单 5.1 显示了适合于创建 Telnet 客户端的基类（清单 5.2 是改动了这个类，除了做客户端，同时可以用做服务器）。使用这个对象处理协议上的要求，会较容易地创建 Telnet 客户端应用，只需简单地从 TelnetTTY 对象继承衍生出一个子类。

子类能为处理接收到的字符提供一个 processChar 方法。而方法 send 允许子类发送字符到远程机器。服务器名（可选）以及端口号传给基类的构造函数做参数。

清单 5.5 是将所有收到的字符变换成大写的一个简易的子类。main 函数简单地从系统控制台那里读到输入，并将它直接发送到远程主机那里。

清单 5.5 这个 Telnet 客户端将所有收到的字符变换成大写

```
import java.io.*;

public class TelnetTest extends TelnetTTY {
    public TelnetTest(String host) throws IOException {
        super(host);
    }
}
```

```

• public synchronized void processChar(int c) {
    char cu=Character.toUpperCase((char)c);
    System.out.print(cu);
}
public static void main(String args[]) throws Exception {
    int c;
    TelnetTest tty=new TelnetTest(args[0]);
    while ((c=System.in.read())!=-1) {
        if (tty.getErrorFlag()) break;
        tty.send(c);
    }
}
}

```

5.2.8 从基类创建一个 Telnet 服务器端

因为 Telnet 协议非常对称，对 TelnetTTY 类（清单 5.1）的少许修改就可以让你创建一个客户端或者服务器应用。更新代码列在清单 5.2 里。

创建一个新服务器很简单，做法如下：

1. 重载 TelnetTTY。
2. 为接收套接字的对象提供一个构造函数。

另外，可以重载方法 `connected` 来处理在客户端创建连接时想要执行的任何任务，重载方法 `processChar` 处理接收到的字符。

清单 5.6 显示了一个 Telnet 服务器程序，当客户端连接时，它打印一个回文，接着强制连接关闭。可以在深入介绍一节里找到另一个 Telnet 服务器程序（见清单 5.3）。

清单 5.6 这个 Telnet 服务器程序也使用了类 TelnetTTY

```

import java.io.*;
import java.net.Socket;

public class TServer extends TelnetTTY {
    public TServer(Socket s) throws IOException {
        super(s);
    }
    public void connected() {
        try {
            send("A man, a plan, a canal, Panama!\n");
            sock.close();
            stopthread=true;
        }
        catch (IOException e) { }
    }

    public static void main(String args[]) throws Exception {

```



```
TelnetTTYServer(TServer.class);  
}  
  
}
```

5.2.9 使用 TelnetWrapper

Telnet Applet (小程序) 开放源码 (在 <http://www.mud.de/se/jta/>) 包含一个类, 用它可以很容易地编写 Telnet 客户端程序。类 TelnetWrapper 与类 TelnetTTY 类似, 但它还有一个特征, 要你等待来自远程计算机输出的一定数目的字符。

在清单 5.7 里, 会看到使用类 TelnetWrapper 的一个简单的 Telnet 客户端, 方法 connect 指定了机器名和端口, 方法 login 提供了用户名和密码, 应用的说明部分发生在程序调用 setPrompt 方法的时候, 现在 TelnetWrapper 对象知道在这个串成为某个命令 (假设来源于 Unix 的 shell) 的输出之前产生数据。调用 waitFor 方法会等待主机查询有关终端的类型。在程序响应之后 (使用 send), 它发出一个 df 命令 (检查磁盘空间)。因为前一个 setPrompt 调用, send 命令将返回 df 命令的输出。

清单 5.7 类 TelnetWrapper 使得编写等待主机响应的 Telnet 客户端程序比较简单

```
import de.mud.telnet.*;  
public class JtaScript {  
    public static void main(String args[]) {  
        TelnetWrapper telnet = new TelnetWrapper();  
        String username="alw";  
        String hostname="colossus";  
        try {  
            telnet.connect(hostname, 23);  
            telnet.login(username, "password");  
            telnet.setPrompt("/users/" + username + " Ñ>");  
            telnet.waitFor("TERM =");  
            telnet.send("dumb");  
            System.out.println(telnet.send("df"));  
        } catch (java.io.IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

使用类 TelnetWrapper, 步骤如下:

1. 实例化一个 TelnetWrapper 对象。
2. 调用 connect 指定远程主机及其端口。
3. 调用 login 确定用户名及密码。
4. 使用 waitFor、setprompt 和 send 与主机交互。
5. 可以使用 disconnect 中断与主机的连接。

第6章 FTP 协议

6.1 深入介绍

我的屋子里放着最后六台 PC，当然，我的计算机在我家里的办公室里。我有一台旧的便携式电脑不值得卖，一台新一点的便携式电脑与我的台式机几乎速度一样快。我的两个儿子的房间里都有电脑。最后，外边还有一台旧式电脑在我的电子实验室（实际上是我的车库）。

在有两台电脑之后不久，我就在顶楼铺好网络主体，并设置好了小网络。但是，由于某些原因，在车库上边铺不了电缆。不变的是，我碰巧需要的文件要么当我在车库时它在网络上，或者当我在一台网上计算机那儿时，它在车库。我认为那是计算机变化的欺骗规则。

直到最后我将一些电缆铺到我的代用实验室，使用所谓的 Nike 网来传输文件（你知道，穿耐克鞋在礼堂里滑倒，会带走一层软布）。我想起在 Internet 出现之前，人们对大规模有着相同的经历，我还记得属于 Hewlett Packard 计算机小组的用户会随邮件邮寄磁带。你将它随意保存多少天都可以，到最后可以加上你有的东西，将它邮到另一个站点。

那就是为何 FTP（文件传输协议）成为最早的 Internet 标准之一的原因。它的任务是允许计算机（有时与计算机不相同）传输文件。像很多 Internet 协议一样，交互对人是很 useful 的，但它实际上是为了机器而考虑的。

有些方面，FTP 使用了与 Telnet 相同的思想。你期望在 FTP 里能看到像 NVT（网络虚拟终端）和其他一些 Telnet 协议选项一样的内容。但是，Telnet 试图连接终端使用不同模式的地方，FTP 考虑得更多的是文件的不同。在当今相对均匀化的世界里，可以认为文件是一大块字节流。但是，有些计算机使用记录、奇怪的字长、或者独特的数据结构来存储文件，甚至有些现在的计算机如 Macintosh，它的文件很奇怪，不能翻译成简单的字节串（每个 Mac 文件实际上有两个子文件即所谓的二叉文件）。

6.1.1 基础

FTP 是一种奇特的协议，因为它使用两种套接字，主套接字（TCP 端口 21）处理从客户端发送至服务器的命令，同时也处理来自服务器的响应；其余端口，缺省为端口 20（可以改变），处理的是数据。

相对现在的 Internet 而言，这种机制可能显得有些奇怪。但是，控制端口没有必要处理二进制数据。另外，尽管很少有程序支持这种功能，理论上你还是可以使用控制端口来建立服务器与第三方机器的数据传输。

如果看了 RFC959 (FTP 的基本规范), 会发现里边有大量的各种字符编码与存储表示方法之间的转换内容。例如, RFC 里提到, DEC TOPS-20 使用 5 个 7 位 (bit) 字符存储 ASCII 文本, 使用左校验、36 位字长的 IBM 大型机, 使用扩展的 BCD 码 (EBCDIC)。有些计算机并不把文件表示为字节流。这个 RFC 文档包含了所有这些细节。

幸运的是, 在当代世界, 这种转换很多已经过时了, 即使你使用的计算机处理文件和字符的方式比较奇怪, Internet 也已经强制绝大多数平台互相作出让步, 因此现在没有太大的不同。

你可能想知道为何要关心这些细节。假设你在远程的服务器里有一张你假期的 JPEG 文件以及描述文件里的图片的文本文件, 如果要下载 JPEG 文件, 当然期望所有字节都完整的通过网络。但是, 如果要将一个 ASCII 文本文件装载进 IBM 大型机里, 就不容易读出来。那就是为什么 FTP 区分包含文本的文件与应该逐字拷贝的文件的原因。

实际上, 用户要区分这些文件类型的最大原因是因为 MS-DOS 和 Windows 在文件的行末使用两个字符 (回车符和换行符), 而 Unix 和大多数其他系统只用一个换行符。大多数 FTP 服务器会对文本模式中的行末进行合适的修改, 但不会修改二进制文件。

另有一个不常用的选项, 允许用户指定文本文件中打印控制码的类型。假设, 你可以使用 FTP 将文件从服务器发送到一台打印机那里, 打印机监听某远程机器的一个套接字。现在的 FTP 应用很少去实现处理打印控制码。

6.1.2 传输

当用户请求发送或者接收文件时, 服务器一般会打开一个针对用户机器的套接字, 这意味着每个 FTP 客户端在某种意义上也可以成为服务器。缺省情况下, 数据套接字为套接字 20, 但一般情况下, 客户端会选择其他端口号 (特别是因为在 20 端口每次只有一个用户可以监听)。

当用户请求到一批文件的列表时, 服务器使用特殊的数据连接来传送结果。服务器可以使用两种方式显示这个列表: 一种方式是每行都是一个文件名, 逐行显示; 另一种方式是显示完整的清单, 可以包含能获取的文件大小以及日期时间等详细信息。人们喜欢看到完整的列表, 但是没有标准的格式, 因此, 程序很可能更愿意用没有格式的列表。

有些 FTP 客户端不能充当数据套接字的服务器。特别是当客户端受防火墙保护时。防火墙会拒绝收到的连接请求, 这样会让客户端不能使用 FTP。

为了补救这一点, 可以请求服务器使用被动 (passive) 模式, 使用被动模式, 服务器创建一个独立的数据套接字并通知客户端它用于数据连接的端口号。接着服务器就在此端口上监听来自客户端的连接。

通常 FTP 程序使用缺省的流 (stream) 模式。你也可以选择块 (block) 模式 (可以发送数据, 附带一个指示块长的题头) 或者是压缩模式。压缩模式使用行程编码的方式。当然, 如果你真的需要压缩, 那最好是在通过 FTP 发送数据之前, 使用更高效的文件编码方式 (例如 zip)。

顺便说一句，FTP 程序并不执行任何的错误检测或者纠错，因为它们要使用 TCP 端口。底层网络协议应该防止数据错误导致数据混乱的问题。

6.1.3 响应

像我前面提到的一样，FTP 创建出来，对程序的使用是很有用的，但对人来说，还是可理解的。这在服务器控制连接发送相应的命令时特别明显。

响应的前四个字符非常重要。前三个是专用码，表示服务器的具体响应。第四个字符是一个空格，如果只有一行响应文本或者这一行是多行响应的最后一行。如果第四个字符是一个连字符，那么后边就多行响应（它们都有相同的 3 位码）。

这四个特殊字符后边的剩余部分可以包含任何需要为响应所作的描述文本。在大多数情况下，这些只对用户有意义。但是，也有一些响应里包含一些信息，客户端程序需要对它们进行解释。

数字编码的范围通常在 100 到 599 之间，每一位都有特殊的意义，客户端可以用它来指导自己的工作。第一位为客户端指出了成功或者失败的粗略思想：

- 100 至 199——肯定的初步响应。这意味着服务器承认命令的接收，并且，认为命令都是正确的。但是，客户端在进一步发送命令之前，必须继续等待另一个响应。
- 200 至 299——请求的行动完全而且成功。
- 300 至 399——肯定的立即响应。该范围的任何值都表示服务器接收了命令，但需要来自客户端的更多信息才能继续。
- 400 至 499——该范围的码值表示产生了失败，但失败可能是暂时的。例如，假设服务器限于当前的用户数不能打开更多的套接字，如果在此以后试图使用完全相同的传输，它就可能成功。
- 500 至 599——这些码值表示一个永久错误，管理方式相同的命令只会产生另一个错误。

第二位进一步缩小了消息意义的范围，如果消息的第二位是 0，服务器就会对命令的语法感到疑惑；如果是 1，则表示一条信息消息；如果是 2，则表示有关控制和数据连接的响应（也就是套接字相关消息）。与用户登录相关的消息第二位值为 3，文件系统消息第二位值为 5。RFC 没有为第二位值为 4 定义任何消息。

第三位在前两位的限制下指定了更精确的消息。可以在表 6.1 中找到常用消息。标有固定格式的入口在文本区域里必须符合特定的格式。

你将会注意到有些错误码（如 450 和 550）看起来似乎是一样的。记住第一位决定了错误是暂时的还是永久的。例如，如果试图对一个别人已经打开的文件执行操作，那将会产生码值为 450 的响应。你终究还可再来一次，这次可能不会出错，但是如果那个用户不能访问该文件，响应值为 550 将更为合适。

表 6.1 FTP 服务器为每一操作请求作出的响应及其码值

码值	含义	固定格式
110	重启标识响应	是
120	服务将要就绪	否
125	数据连接已经打开	否
150	文件状态良好，将打开数据连接	否
200	命令就绪	否
202	命令没有实现因为本服务器不需要使用它	否
211	系统状态或帮助响应	否
212	目录状态	否
213	文件状态	否
214	帮助	否
215	系统类型（例如，Unix）	是
220	登录服务准备就绪	否
221	服务关闭控制连接	否
225	数据连接打开；没有传输在进行	否
226	关闭数据连接（成功）	否
227	输入被动模式	是
230	用户已经登录	否
250	请求文件操作完成	否
257	路径已创建	否
331	用户名已确认，需要密码	否
332	需要帐号信息	否
350	请求操作推迟，需要进一步信息	否
421	不能获得服务	否
425	不能打开数据连接	否
426	连接关闭（放弃）	否
450	文件不可得	否
451	由于局部错误而放弃	否
452	由于缺乏空间而放弃	否
500	命令的语法错误	否

续表

码值	含义	固定格式
501	参数的语法错误	否
502	命令没有实现	否
503	错误的命令序列	否
504	带有这个参数的命令没有实现	否
530	没有登录成功	否
550	因为文件不可得而失败	否
551	因为页类型未知而失败	否
552	因为存储分配超过限制而失败	否
553	因为错误文件名而失败	否

6.1.4 登录

FTP 服务器通常需要用户对他们自己进行鉴别。关于服务器如何验证用户 ID 和密码,其细节当然与具体的系统相关(除非你使用自己私有的鉴别方案,例如在一个属性文件里列出所有用户)。

当客户端创建一个到服务器的初始连接时,服务器通常发送 220 响应码来表示它已准备就绪。客户端应该使用 USER 命令来响应,这个命令包含用户的 ID。接着客户端会收到 331 这个响应码,第一位值为 3,所以你知道服务器期待着更多的信息,那个信息就是密码(使用 PASS 命令来发送)。一旦服务器接受了用户 ID 和密码,它将使用 230 响应码来响应。

有些系统也要求帐号要么用于登录,要么用于特定的操作。客户端可以使用 ACCT 命令在任何时候指定它,如果服务器端需要帐号信息,它可以发送 332 响应码到客户端。

6.1.5 创建连接

尽管管理控制连接与创建任何其他套接字连接是相同的,数据连接却截然不同。数据连接在客户端和服务端之间对话的过程中,可能在不同的时刻出现和消失。

通常,服务器希望客户端监听 20 端口。由于这对大多数客户端来说,并不常用,客户端常见的做法是发送 PORT 命令。该命令需要 6 个 10 进制数(以 ASCII 形式表示)。前四个数是客户端的 IP 地址,最后两个数表示 16 位的端口号。第一个数是最重要的 8 位二进制数,最后一个数意义最小。这个命令要求将这些数以逗号隔开。

如果客户端 IP 是 10.1.1.5, 监听 513 端口,那么 PORT 命令将可以表示如下:

```
PORT 10,1,1,5,2,1
```

有趣的是,IP 地址不一定要与客户端的地址相匹配。使用 PORT 命令指导服务器与第三

台计算机之间发送或者接收数据也是合法的。当然，第三台计算机将不得不监听具体的端口号，这在现在的使用当中不太常见。

在 Java 程序中形成 PORT 命令需要从控制套接字中获得 IP 地址和端口号，进行重新格式化（见表 6.1）。方法 `getLocalAddress`（`InetAddress` 的成员）返回一个 `InetAddress` 对象，接着可以调用 `getHostAddress` 找到用于和 FTP 服务器对话的本地地址。但是，你必须要将所有的点号转换成逗号，以符合命令 PORT 的语法。将 IP 地址放入 `StringBuffer`，将很容易完成该任务。最后，可以调用 `append` 来插入端口号（不要忘记先将它拆成 2 个字节）。

在清单 6.1 中，在命令 PORT 后的代码后是 LIST 命令，请求服务器在数据端口上发送可得到的文件的列表，接着程序开始监听 `ServerSocket`（使用 `accept`）。

清单 6.1 这段 Java 代码构成一个 PORT 命令

```
// ctlskt is the FTP control socket
// datskt is the FTP data socket
ServerSocket datsktsvr = new ServerSocket(0); // pick a port
StringBuffer cmd = new StringBuffer("PORT ");
cmd.append(ctlskt.getLocalAddress().getHostAddress());
for (int i = 0; i < cmd.length(); i++) {
    if ('.' == cmd.charAt(i))
        cmd.setCharAt(i, ',');
}
cmd.append(",");
cmd.append(Integer.toString(port/256));
cmd.append(",");
cmd.append(Integer.toString(port&0xFF));
status=sendCmd(cmd.toString());
if (status!=200) return -1;
status=sendCmd("LIST");
// wait for remote connection (could time out)
datskt=datsktsvr.accept();
```

当然，PORT 命令假设你想要服务器与客户端间建立连接。如果想连接到服务器，可以发送 PASV 命令。如果服务器支持被动模式，这将与码值 227 相对应。其响应也会有 IP 地址和端口，使用与 PORT 命令相同的格式，它将嵌入到返回的串中。例如，下面是来自 Internet 上一个真实的 FTP 服务器的响应：

```
227 Entering Passive Mode (208,146,45,13,17,47)
```

接着客户端需要将这个串分离成正确的参数。可以在清单 6.2 中找到这个例子，这里，变量 `lastResponse` 里有从服务器对 `sendCmd` 调用的响应。这里，程序要将逗号替换为点号，重新组织端口号，接着构造一个 `Socket` 对象。

被动模式在大多数防火墙下能够工作。少量服务器不支持被动模式，因此一种明智的方法是试用被动模式，如果会返回错误，可以试着发送 PORT 命令（RFC1579 推荐这种方法）。

清单 6.2 这段代码发出 PASV 命令，以初始化被动模式

```

status=sendCmd("PASV");
if (status<200||status>299) return -1;
// find IP/port
for (i=4;i<lastResponse.length();i++)
if (Character.isDigit(lastResponse.charAt(i))) break;
if (i==lastResponse.length()) return -1; // not found
token=new java.util.StringTokenizer(lastResponse.substring(i),",");
if (token.countTokens()!=6) return -1; // unknown IP/port
for (i=0;i<4;i++) {
IP.append(token.nextToken());
if (i!=3) IP.append(".");
}
port=Integer.parseInt(token.nextToken())*256;
port+=Integer.parseInt(token.nextToken());
datskt=new Socket(IP.toString(),port);
status=sendCmd("LIST");

```

一旦你拥有 `datskt`，不管是从调用 `accept` 得到（常规模式）还是从 `Socket` 的构造函数（被动模式）得到，都可以容易地发送数据到套接字（或者从套接字那里读数据）。在清单 6.3 中可以找到相关代码，从套接字中读数据，在变量 `w`（一个文件）中向 `writer` 写数据。

清单 6.3 使用类 `stream` 或者类 `writer` 向数据套接字中写数据比较容易

```

datReader=new InputStreamReader(datskt.getInputStream());
// transfer data until socket closes
int ch;
do {
ch=datReader.read();
if (ch!=-1) w.write((char)ch);
} while (ch!=-1);
w.flush();
datskt.close();

```

6.1.6 FTP 命令细节

RFC 定义了 FTP 服务器可能实现的很多命令。当然，你可以自由地抛弃很多未实现的命令。你必须为那些你不想支持的命令返回恰当的错误响应码。但有很多客户端假设会有特定的命令可以得到（如 `USER`、`PORT`、`LIST` 和 `NLIST`）。很多客户端不能很好处理那些不支持的命令。

表 6.2 显示了那些可用的命令。其中一些在现在的程序中使用不多。如果你想亲手使用这些命令做试验，总可以用 `Telnet` 程序连接到一个 FTP 服务器（确定使用端口 21）。当然，你不能打开数据端口，因此你也不能使用需要数据连接的那些命令，诸如 `RETR`、`LIST` 或者其他数据连接相关命令。

很多命令行方式的 FTP 客户端使用类似的命令。因此当键入 SITE 或者 PWD 时, 客户端会将命令发送到服务器那里。很多客户端也支持 QUOTE 命令, 这样你可以向服务器直接发送任何命令。因此键入 HELP 可能会为本地 FTP 程序显示帮助信息, 但键入 QUOTE HELP 将会显示服务器的帮助信息。

表 6.2 FTP RFC 指定的 FTP 命令

命令	参数	说明	可能的初始响应
USER	用户名	指定用户名	230, 530, 500, 501, 421, 331, 332
PASS	密码	指定密码	230, 202, 530, 500, 501, 503, 421, 332
ACCT	帐号信息	指定用户帐号	230, 202, 530, 500, 501, 503, 421
CWD	目录名	改变工作目录	250, 500, 501, 502, 421, 530, 550
CDUP	空	改变到父目录 (与很多系统中的 CWD 目录相同)	250, 500, 501, 502, 421, 530, 550
SMNT	Mount 信息	结构 mount (改变文件系统——如, mount 一个磁带或者可移动磁盘)	202, 250, 500, 501, 502, 421, 530, 550
REIN	空	重新初始化, 需要新的 USER 命令来继续	120, 220, 421, 500, 502
QUIT	空	结束会话	221, 550
PORT	我 IP 地址以及端口号	指定数据端口	200, 500, 501, 421, 530
PASV	空	请求被动模式	227, 500, 501, 502, 421, 530
TYPE	文件类型代码	选择 ASCII, EBCDIC 或者 Image (镜像) (A, E, I) 文件类型	200, 500, 501, 504, 421, 530
STRU	结构代码	选择文件结构类型 (缺省情况下, F 用于代表 File)	200, 500, 501, 504, 421, 530
MODE	模式代码	选择传输模式 (缺省情况下, 使用 S 来表示 Stream)	200, 500, 501, 504, 421, 530
RETR	文件名	取回文件	125, 150, 450, 550, 500, 501, 421, 530
STOR	文件名	存储文件	125, 150, 532, 450, 452, 553, 500, 501, 421, 530

续表

命令	参数	说明	可能的初始响应
STOU	空	作用特定的文件名存储, 服务器的响应将含有该文件名	125, 150, 532, 450, 452, 553, 500, 501, 421, 530
APPE	文件名	如果指定文件不存在, 就与 STOR 命令相同。但是, 如果文件存在, 当前文件会添加到存在的文件末尾 (追加模式)	125, 150, 532, 450, 452, 553, 500, 501, 421, 530
ALLO	大小	为下一个 STOR、STOU 或者 APPE 命令分配指定数目字节的空间。能常只在需要文件分配的机器上实现此命令	200, 202, 500, 501, 504, 421, 530
REST	标识	通知服务器下一次传输应该在检查点重启 (由标识参数来指定)	500, 501, 502, 421, 530, 350
RNFR	文件名	指定你想重命名的文件的当前文件名: 总是跟随着 RNT0 命令	450, 550, 500, 501, 502, 421, 530, 350
RNT0	文件名	指定想重命名的文件的新文件名: 它的前边总是 RNFR 命令	250, 532, 553, 500, 501, 502, 503, 421, 530
ABOR	空	放弃当前传输 (通常发送紧急 Telnet 消息)	225, 226, 500, 501, 502, 421
DELE	文件名	删除一个文件	250, 450, 550, 500, 501, 502, 421, 530
RMD	目录名	删除一个目录	250, 500, 501, 502, 421, 530, 550
MKD	目录名	创建一个目录	257, 500, 501, 502, 421, 530, 550
PWD	空	输出当前目录	257, 500, 501, 502, 421, 550
LIST	路径名	传输一个对人可读的指定文件或者目录下的文件列表: 如果没有参数, 则使用当前目录	125, 150, 450, 500, 501, 502, 421, 530
NIST	路径名	与 LIST 类似, 该命令只传输文件名列表, 每行一个文件名	125, 150, 450, 500, 501, 502, 421, 530
SITE	命令	发送站点指定命令 (如, 很多 Unix 的 FTP 服务器使用该命令允许你对文件执行 chmod 命令)	200, 202, 500, 501, 530
SYST	空	返回系统类型的标准名, 缺省情况下可能是一个字节大小	215, 500, 501, 502, 421

续表

命令	参数	说明	可能的初始响应
HELP	命令	返回关于指定命令的帮助文本（或者是如果没有参数的情况下返回它的梗概）	211, 214, 500, 501, 502, 421
NOOP	空	没有操作	200, 500, 421

6.1.7 考虑客户端

如果理解基础的 FTP 协议，那么写一个客户端程序不是太难。在本章的快速解决方案节里，你会看到一个简单的客户端程序，按照你的选择显示服务器上的目录列表。因为目录列表和文件传输多少有些相同，你甚至可以写一些函数，覆盖所有常需的功能。

特别地，一个典型的设计将包含下述方法：

- **connect**——与服务器相连，并提供用户名和密码。
- **dataSocketIn**——打开数据套接字并将数据读到另一个流里（很可能是一个文件）。
- **dataSocketOut**——打开数据套接字并将数据写到另一个流里。
- **dataSocket**——创建数据套接字或者是当使用被动模式时与服务器的数据套接字相连；常用在 **dataSocketIn** 和 **dataSocketOut** 中。
- **getResponse**——从服务器那里获取响应码，并考虑到可能出现多行响应。
- **sendCmd**——发送命令到服务器，并等待响应。

使用这些方法，使用 FTP 命令来测试服务器就比较简单了。**dataSocketIn** 命令特别有意义。向它传递一个命令，将导致数据传输。它使用 **dataSocket** 处理数据连接，接着会将来自套接字的数据传到你选择的流中。可以在清单 6.4 中找到 **dataSocketIn**。**dataSocketOut** 方法很相似，但你也可以料到，它是从你选择的流中读数据，接着将它发送到数据套接字中。

清单 6.4 方法将来自 FTP 服务器的数据传送到你选择的流中

```
public int dataSocketIn(String datCmd,Writer w) throws IOException {
    int status;
    status=dataSocket(datCmd);
    if (status!=0) return status;
    datReader=new InputStreamReader(datskt.getInputStream());
    // transfer data until socket closes
    int ch;
    do {
        ch=datReader.read();
        if (ch!=-1) w.write((char)ch);
    } while (ch!=-1);
    w.flush();
    datskt.close();
}
```

```
return getResponse();  
}
```

6.1.8 考虑服务器端

写 FTP 服务器端程序要求监听控制连接，并对接收到的命令行进行解析分离。这要求使用某些方法来读命令行，并进行各种方式的解析。当然，你可以只用 String 里简单的方法来跳过空格，将输入的第一个词隔离，接着用已知的词来匹配命令行里的词，直到发现匹配的结果，稍微复杂的方法是使用类 StringTokenizer 来隔离输入的第一个词。

但无论是使用哪一种方法，都会出现很多 if 分支语句，而它将为增加更多子项产生麻烦，很多服务器会有相同的问题，因此决定试图创建可重用的类来使 FTP 服务器端程序变得易写一些。

我写的第一个类是 LineServer（见清单 6.5）。这个类，从类 MTServerBase 继承而来，处理服务器期待命令行输入的各种值产生的分支，它不解决命令的解析问题，只给一个使用的合理框架。

要使用 LineServer，你要扩展自己的类，并重载 process 和 init。服务器在你接收每行命令时会调用 process 方法，如果你想终止与当前客户端的连接，可以使用这个方法返回 false。当客户端连接时，调用 init。这样就可以发送任何需要的消息来初始化客户端了。

清单 6.5 类 LineServer 对任何逐行读命令行输入的服务器都有用

```
// Line server -- generic line-oriented server  
// Al Williams  
import java.io.*;  
import java.net.*;  
  
public class LineServer extends MTServerBase  
{  
    protected BufferedReader iread;  
    protected BufferedWriter owrite;  
    public boolean running=false;  
    public void run() {  
        try {  
            iread=new BufferedReader(new  
                InputStreamReader(socket.getInputStream()));  
            owrite=new BufferedWriter(new  
                OutputStreamWriter(socket.getOutputStream()));  
            if (!init()) {  
                socket.close();  
                return;  
            }  
            running=true;  
            while (true) {  
                String line = iread.readLine();
```

```

        if (!process(line)) {
            running=false;
            socket.close();
            return;
        }
    }
    catch (IOException ioe) {
        running=false;
    }

}

// You'll override this (and provide a main that calls startServer)
public boolean process(String line) throws IOException {
// this line is here just so we can throw IOExceptions
// from subclasses
    if (line==null) throw new IOException();
    return false; // return true to end connection
}

// You'll override this if you want any custom start processing
public boolean init() throws IOException {
    if (running) throw new IOException(); // just for subclass
    return true; // false to abort
}
}

```

在我使用 C 编程的日子里，我可能使用一张简单的表来解决命令解析问题，这个表将匹配串和函数指针，因此当串与表的入口相匹配时，你可以直接调用函数。但是 Java 没有函数指针，对吗？

当然，Java 确实没有函数指针，但在这种情况下，它有一个 Method 对象可以做几乎相同的事情。Method 对象是 Java 对象用来在运行时了解其他类（或者它们自己）的反射 API 的一部分。我决定以简单的方法使用 Method 来将命令名映射成方法。

这就出现了清单 6.6 中的类。每个 ParseTableEntry 的实例包含一个 String（在 word 域）和一个 Method 对象（在 func 域，两个域都是私有的）。构造函数有三个参数：匹配词、拥有要调用方法的类和方法名。

你可以调用两个方法来使 ParseTableEntry 变得有用。首先，可以调用 equals 方法。如你所料，这将试图让入口的词与串相匹配，在我写这个类的方法里，这个匹配是对大小写不敏感的，但是你可以很容易地修改它（甚至让它成为可选的）。

第二个你要调用的方法是 call 方法。这段代码封装了实际上调用这个方法的繁重的工作。第一个参数是你需要用于这个调用的实例。另两个参数是你指定的方法（String 和 StringTokenizer）的参数。当然，那意味着你调用的方法必须要用到那些参数，它也应该返回一个 boolean 型的值，这也是 call 方法要返回的。

当然，只使用一个 `ParseTableEntry` 对象意义不大。相反，你需要一组这种对象，该对象有一个静态方法（`parse`）接收一个数组、一个对象实例、待解析的词、你正检查的命令行以及用于获取词的 `StringTokenizer` 实例。这个方法查询那张表直到发现相匹配的一个入口或者空串入口（匹配任何词并且是最后一个入口）。它会为你创建一个调用，并返回要返回的值。

清单 6.6 类允许你建立数组来映射命令名和要调用的方法

```
// Helper class for SFTPServer
// or any program that needs a simple first word parse
import java.lang.reflect.*;
import java.util.StringTokenizer;

public class ParseTableEntry {
    private String word;
    private Method func;

    public static boolean parse(ParseTableEntry [] parseTable,
                                Object othis,
                                String word, String line,
                                StringTokenizer token) {
        for (int i=0; i<parseTable.length; i++) {
            if (parseTable[i].equals(word)
                || parseTable[i].equals(""))
                return parseTable[i].call(othis, line, token);
        }
        // should never get here unless no "" case in table
        return false;
    }

    public ParseTableEntry(String word, Class c, String methd) {
        Class [] cary = new Class[2];
        this.word=word;
        cary[0]=String.class;
        cary[1]=StringTokenizer.class;
        try {
            func=c.getMethod(methd, cary);
        }
        catch (Exception e) {
            System.err.println("ParseTable: " + e);
        }
    }

    public boolean equals(String s) {
        return word.compareToIgnoreCase(s)==0;
    }

    public boolean call(Object thisptr, String line, StringTokenizer token) {
        Object arg[] = new Object[2];
        arg[0]=(Object)line;
        arg[1]=(Object)token;
        try {
            Boolean bv;
            bv=(Boolean)func.invoke(thisptr, arg);
        }
    }
}
```

```

        return bv.booleanValue();
    }
    catch (Exception e) {
        System.err.println("Invoke: " + e);
    }
    return false; // error
}
}

```

这个简单的类通过隐藏所有的细节，使得使用 `Method` 这个类不会太麻烦。尤其是，你将注意到你的类不得不在 `Object` 数组里包含方法的参数，接着从 `Boolean` 型对象中挑出 `boolean` 型返回值，这不会太难，但很麻烦并且看起来很杂乱，为何不将它们封装成一个类呢？

调用程序只是建立一个入口数组，如下所示：

```

ParseTableEntry parseTable[] = {
    new ParseTableEntry("LIST", SFtpServer.class, "listCmd"),
    new ParseTableEntry("NLST", SFtpServer.class, "listCmd"),
    // always make this the last entry
    new ParseTableEntry("", SFtpServer.class, "unknownCmd")
};

```

运用这个方法建立一个 FTP 服务器相对容易些，只用简单地创建一个 `LineServer` 子类。在 `process` 例程里，可以使用一个 `ParseTableEntry` 对象数组跳转到你的类里的正确的例程里。在清单 6.7 中会找到这个例子。

清单 6.7 中的服务器 (`SFtpServer`) 不是很复杂，然而它可能还要与大多数 FTP 客户端交互。服务器只为两个文件服务，一个是 `a.txt`，另一个为 `b.txt`。实际上程序读它自己的源码，接着发送它，尽管你可以使用另一个文件或者改变程序，来让它使用真实的文件系统工作。

因为服务器实际上并不是为真实的文件系统服务，它没有必要处理目录、文本模式或者任何其他系统相关问题。用户不能存储文件，并且所有登录进去的用户都当作是匿名用户。

清单 6.7 这个简单的 FTP 服务器端程序解释了使用本章里的一些工具可以很容易地创建一个服务器

```

// Simple FTP server
import java.io.*;
import java.net.*;
import java.util.*;

// **** MAIN CLASS
public class SFtpServer extends LineServer {

    private ParseTableEntry parseTable[] = {
        new ParseTableEntry("USER", SFtpServer.class, "userCmd"),
        new ParseTableEntry("QUIT", SFtpServer.class, "quitCmd"),
        new ParseTableEntry("PASS", SFtpServer.class, "passCmd"),
        new ParseTableEntry("ACCT", SFtpServer.class, "unimplCmd"),
        new ParseTableEntry("SMNT", SFtpServer.class, "unimplCmd"),
        new ParseTableEntry("CWD", SFtpServer.class, "unimplCmd"),
    };
}

```



```

    new ParseTableEntry("CDUP", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("REIN", SftpServer.class, "reinCmd"),
    new ParseTableEntry("PASV", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("STRU", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("MODE", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("TYPE", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("STOU", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("APPE", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("ALLO", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("REST", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("RMD", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("MKD", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("PWD", SftpServer.class, "pwdCmd"),
    new ParseTableEntry("SITE", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("SYST", SftpServer.class, "systCmd"),
    new ParseTableEntry("HELP", SftpServer.class, "helpCmd"),
    new ParseTableEntry("NOOP", SftpServer.class, "noopCmd"),
    new ParseTableEntry("STAT", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("RNFR", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("RNT0", SftpServer.class, "unimplCmd"),
    new ParseTableEntry("DELE", SftpServer.class, "unimplCmd"),

    new ParseTableEntry("PORT", SftpServer.class, "portCmd"),
    new ParseTableEntry("RETR", SftpServer.class, "retrCmd"),
    new ParseTableEntry("LIST", SftpServer.class, "listCmd"),
    new ParseTableEntry("NLST", SftpServer.class, "listCmd"),
// always make this the last entry
    new ParseTableEntry("", SftpServer.class, "unknownCmd")
};

int loginstate=0; // 0=none, 1=user OK, 2= user/pw ok, 3=user Not OK
String dataportip;
int dataport=20;

public void writeString(String s) throws IOException {
    owrite.write(s,0,s.length());
    owrite.write("\r\n");
    owrite.flush();
}

public boolean init() throws IOException {
    // set default initial IP
    dataportip=socket.getInetAddress().getHostAddress();
    writeString("220-Sftp. Anonymous logins accepted");
    writeString("220-Your IP: " + dataportip);
    writeString("220-Default data port: " + dataport);
    writeString("220 Server ready");
    return true;
}

public boolean process(String line) throws IOException {
    boolean rv=false;

```

```
StringTokenizer token=new StringTokenizer(line);
String word=null;
try {
    word=token.nextToken();
}
catch (NoSuchElementException nsee) {} // just a blank line
if (word==null) return true; // ignore blank lines
String output;
rv=ParseTableEntry.parse(parseTable,this,word,line,token);
owrite.flush();
return rv;
}

// getMethod only works on public methods
public boolean unknownCmd(String line,StringTokenizer token)
throws IOException {
    writeString("500 Unknown command ");
    return true;
}

public boolean quitCmd(String line,StringTokenizer token)
throws IOException {
    writeString("221 Goodbye");
    return false;
}

public boolean unimplCmd(String line,StringTokenizer token)
throws IOException {
    writeString("202 Command not implemented.");
    return true;
}

public boolean pwdCmd(String line,StringTokenizer token)
throws IOException {
    if (loginstate!=2) return loginerr();
    writeString("257 \"/\" is the current working directory.");
    return true;
}

public boolean loginerr() throws IOException {
    writeString("530 Please login");
    return true;
}

public boolean reinCmd(String line,StringTokenizer token)
throws IOException {
    writeString("200 Command OK.");
    loginstate=0;
    return true;
}

public boolean systCmd(String line,StringTokenizer token)
```

```

throws IOException {
    if (loginstate!=2) return loginerr();
    writeString("215 UNIX Type: L8");
    return true;
}

    public boolean passCmd(String line,StringTokenizer token)
throws IOException {
    String pw=token.nextToken();
    if (loginstate==0) {
        writeString("503 Login with USER First");
        return true;
    }
    if (loginstate==1 && pw!=null && !pw.equals("")) {
        System.err.println("Login from " + pw);
        loginstate=2;
        writeString("230 User logged in");
        return true;
    }
    loginstate=0;
    writeString("530 Login incorrect");
    return false;
}

    public boolean userCmd(String line,StringTokenizer token)
throws IOException {
    // we only allow anonymous
    if (token.nextToken().equals("anonymous")) {
        loginstate=1;
    }
    else {
        loginstate=3;
    }
    writeString("331 Send e-mail as password");
    return true;
}

    public boolean noopCmd(String line,StringTokenizer token)
throws IOException {
    writeString("200 Command OK");
    return true;
}

    public boolean portCmd(String line,StringTokenizer token)
throws IOException {
    StringBuffer b=new StringBuffer();
    try {
        for (int i=0;i<4;i++) {
            String tmp=token.nextToken(" ,");
            if (tmp==null) throw new Exception();
            int n=Integer.parseInt(tmp); // throw if not integer
            if (n<0||n>255) throw new Exception();
            b.append(tmp);
        }
    }
}

```

```

        if (i!=3) b.append(".");
    }
    dataportip=b.toString();

    String tmp=token.nextToken(" ,");
    dataport=Integer.parseInt(tmp)*256;
    tmp=token.nextToken();
    dataport+=Integer.parseInt(tmp);
}
catch (Exception e) {
    writeString("501 Illegal Port Command");
    return true;
}
writeString("200 Command OK");
return true;
}

public boolean retrCmd(String line,StringTokenizer token)
throws IOException {
    String fn=token.nextToken();
    if (fn.equals("a.txt")) {
        sendData(new FileReader("SftpServer.java"));
    }
    else if (fn.equals("b.txt")) {
        sendData(new FileReader("ParseTableEntry.java"));
    }
    else {
        writeString("550 No such file");
    }
    return true;
}

public boolean listCmd(String line,StringTokenizer token)
throws IOException {

    sendData(new StringReader("a.txt\r\nb.txt\r\n"));
    return true;
}

public boolean helpCmd(String line,StringTokenizer token)
throws IOException {
    writeString("214-Simple Java Server by alw@al-williams.com");
    writeString("214 No further help available.");
    return true;
}

public boolean sendData(Reader r) throws IOException {
    int c;
    Socket s=null;
    try {

```

```
writeString("150 Opening Data connection");
s=new Socket(dataportip,dataport);
OutputStreamWriter w=new OutputStreamWriter(s.getOutputStream());
do {
    c=r.read();
    if (c!=-1) w.write((char)c);
    } while (c!=-1);
w.flush();
writeString("226 Transfer Complete");
s.close();
return true;
}
catch (Exception e)
writeString("426 Transfer error");
s.close();
return true;
}
}

static public void main(String args[]) {
    SftpServer.startServer(21,SftpServer.class);
}
}
```

6.2 快速解决方案

6.2.1 查找 FTP 规范

FTP 规范主要在 RFC959（也称为 STD0009）里边。这篇文档定义了 FTP 的主要端口。在 RFC2577（安全性）和 RFC2428（IPV6）里你将会找到附加信息。

6.2.2 连接到 FTP 服务器

几乎所有的 FTP 服务器需要某种类型的用户登录和密码，即使某些服务器会接受匿名用户，其字面意思是使用用户名“anonymous”，通常用你的 email 地址作为密码。

下面是连向一个 FTP 服务器的步骤：

1. 等待服务器的 220 响应码，它表示服务器准备接收命令。
2. 发送 USER 命令，指定用户 ID。
3. 等待响应，如果响应码值为 331，发送 PASS 命令指定密码。
4. 如果 USER 或 PASS 的响应是 230，准备发送命令到服务器。任何其他响应表示有问题发生。

清单 6.8 显示了码值中的一位所做的工作（这是从清单 6.9 的客户端代码抽出来的一段）。

清单 6.8 这段代码登录到一个 FTP 服务器

```
do {
    status=getResponse();
} while (status!=220 && status!=-1);
if (status==-1) {
    return false;
}
status=sendCmd("USER " + user);
if (status==331) status=sendCmd("PASS " + pw);
if (status==332) return false; // no ACCT
if (status!=230) return false;
return true;
```

6.2.3 解释 FTP 的响应

每个来自 FTP 服务器的响应都是单行或者多行的形式。每行有三位数的响应码出现在开头。第四个字符将表示后边是否有更多的行。第四个字符如果是连字号，则告诉客户端期望更多的行；如果是空格，则意味着这是服务器将发送的最后一行（或者唯一一行）。多行响应的所有行的起始位置都有相同的码值。

响应码的第一位告诉客户端码值是失败或者是成功，以及客户端是否应该采取进一步的行动。响应码范围从 100 到 599，其值可分为 5 组：

- Group 1——命令成功；等待进一步的响应码。
- Group 2——命令成功并且完成。
- Group 3——命令成功；发送更多信息等待完成。
- Group 4——命令失败；过一段时间重试。
- Group 5——命令失败；永久错误。

第二位进一步缩小消息意义的范围，具体如下：

- Digit 0——命令中有语法错误。
- Digit 1——信息消息。
- Digit 2——套接字相关消息。
- Digit 3——用户登录消息。
- Digit 5——文件系统消息。

你可以在表 6.1 中的 RFC 定义中找到完整的响应码列表。

6.2.4 管理当前目录

你可以使用 CWD 命令来改变服务器的当前目录。使用 CDUP 可以将目录移到当前目录的父目录，这样也能改变目录。命令 PWD 的响应值为当前目录。

6.2.5 读文件目录

连向服务器时用户想知道的第一件事就是什么文件可以得到。LIST 和 NLST 命令以不同格式提供了这方面的信息 (LIST 命令细节比较多, 而 NLST 的输出结果很容易被程序解析)。

这两个命令将它们的输出提供为伪文件的形式。只要客户端能辨别, 这只不过是请求一个特殊文件, 而刚好其输出是指定目录 (或者是当前目录, 如果没有提供目录) 的内容列表。

要读一个目录列表的步骤与接收一个普通文件很类似:

1. 发送 PORT 命令, 指出你用于数据端口的端口。(当然, 如果你正使用被动模式, 那就要发送 PASV 命令。)
2. 等待码值为 200 的响应。
3. 发送一个 LIST 命令。
4. 等待数据套接字上的一个连接和码值为 150 的响应。
5. 在列表结束时, 服务器将发送码值为 226 的响应。

清单 6.9 显示出一个简单的 FTP 客户端的例子, 用于获取目录列表。要在命令行里提供服务器名、用户名和密码。如果愿意也可以提供目录名作为最后一个参数。

函数 connect 处理登录。当程序需要发送命令时, 它使用 sendCmd, 返回状态码, 并将响应的最后一行存到 lastResponse, 但真正关键的是 dataSocketIn, 这个方法设置了一个数据端口 (或者是主动的或者是被动的, 依赖于 passive 的域值), 接着它发出你提供的命令, 并从数据端口上读数据。

清单 6.9 本程序获取目录列表

```
// FTP LS
import java.net.*;
import java.io.*;

public class FtpLS {
    final public static boolean debug=true; // true for debug trace
    protected Socket ctlskt; // control socket
    protected Socket dataSocket; // data socket
    protected Reader dataReader; // IO Stream for data socket
    protected Writer dataWriter;
    protected BufferedReader ctlin; // Control port reader
    protected OutputStream ctliout; // Control port writer
    public String lastResponse; // Last response from host
    public boolean passive = false; // passive mode

    // Connect to host -- we don't do ACCT
    public boolean connect(String host,String user, String pw)
        throws UnknownHostException, IOException {
        int status;
        ctlskt=new Socket(host,21);
        ctlin=new BufferedReader(
```

```

    new InputStreamReader(ctlskt.getInputStream()));
    ctlsout=ctlskt.getOutputStream();
    do {
        status=getResponse();
    } while (status!=220 && status!=-1);
    if (status!=-1) {
        return false;
    }
    status=sendCmd("USER " + user);
    if (status==331) status=sendCmd("PASS " + pw);
    if (status==332) return false; // no ACCT
    if (status!=230) return false;
    return true;
}

// This creates a data socket
protected int dataSocket(String datCmd) throws IOException {
    int port;
    int status;
    ServerSocket datsktsvr;
    if (passive) {
        java.util.StringTokenizer token;
        StringBuffer IP=new StringBuffer();
        int i;
        status=sendCmd("PASV");
        if (status<200||status>299) return -1;
        // find IP/port
        for (i=4;i<lastResponse.length();i++)
            if (Character.isDigit(lastResponse.charAt(i))) break;
        if (i==lastResponse.length()) return -1; // not found
        token=new java.util.StringTokenizer
            (lastResponse.substring(i),",,");
        if (token.countTokens()!=6) return -1; // unknown IP/port
        for (i=0;i<4;i++) {
            IP.append(token.nextToken());
            if (i!=3) IP.append(".");
        }
        port=Integer.parseInt(token.nextToken())*256;
        port+=Integer.parseInt(token.nextToken());
        datskt=new Socket(IP.toString(),port);
        status=sendCmd(datCmd);
    }
    else {
        datsktsvr=new ServerSocket(0);
        port=datsktsvr.getLocalPort();
        //set host/port
        StringBuffer cmd = new StringBuffer("PORT ");
        cmd.append(ctlskt.getLocalAddress().getHostAddress());
        for (int i = 0; i < cmd.length(); i++) {
            if ('.' == cmd.charAt(i))
                cmd.setCharAt(i, ',');
        }
    }
}

```

```

        cmd.append(",");
        cmd.append(Integer.toString(port/256));
        cmd.append(",");
        cmd.append(Integer.toString(port&0xFF));
        status=sendCmd(cmd.toString());
        if (status!=200) return -1;
        status=sendCmd(datCmd);
        // wait for remote connection (could time out)
        datskt=datsocketsvr.accept();
    }
    return 0;
}
// This function pipes everything from the
// data port to the specified writer (which could
// be the console, or a string, or a file)
public int dataSocketIn(String datCmd,Writer w) throws IOException {

    int status;
    status=dataSocket(datCmd);
    if (status!=0) return status;
    datReader=new InputStreamReader(datskt.getInputStream());
    // transfer data until socket closes
    int ch;
    do {
        ch=datReader.read();
        if (ch!=-1) w.write((char)ch);
    } while (ch!=-1);
    w.flush();
    datskt.close();
    return getResponse();
}

// Get a response accounting for possible mutlline responses
protected int getResponse() throws IOException {
    int n;
    String tmp;
    do {
        lastResponse=ctlin.readLine();
        if (debug) System.out.println("DEBUG Received: " + lastResponse);
    } while (lastResponse.charAt(3)=='-');
    tmp=lastResponse.substring(0,3);
    try {
        n=Integer.parseInt(tmp);
    }
    catch (NumberFormatException e) {
        n=-1;
    }
    return n;
}

// Send a command and return status code
public int sendCmd(String cmd) throws IOException {

```

```

        if (debug) System.out.println("DEBUG Sent: " + cmd);
        String tmp=cmd+"\r\n";
        int n;
        ctout.write(tmp.getBytes());
        ctout.flush();
        return getResponse();
    }

    // Main driver

    public static void main(String [] args) throws Exception {
        if (args.length<3||args.length>4) {
            System.out.println(
                "usage: EZftp host user password [directory]");
            System.exit(1);
        }
        FtpLS obj= new FtpLS();
        obj.passive=true;
        if (!obj.connect(args[0],args[1],args[2])) {
            System.out.println("Can't connect");
            System.exit(1);
        }

        // Last argument is directory if present
        if (args.length==4)
            obj.sendCmd("CWD " + args[3]);

        // print directory (chop off response code)
        obj.sendCmd("PWD");
        System.out.println(obj.lastResponse.substring(4));
        // Do a directory
        obj.dataSocketIn("LIST",new OutputStreamWriter(System.out));

        System.exit(0);
    }
}

```

6.2.6 传输文件

传输一个文件与传输一个文件目录基本上完全相同。唯一的不同是你要发出一个 **STOR** 命令以发送文件到服务器,以及发出一个 **RETR** 命令从服务器上获取文件。很明显,使用 **STOR** 命令时,必须在数据连接上提供数据,那要求使用 **dataSocketOut** 方法来补充 **dataSocketIn**。清单 6.10 显示了这样一个方法,你可以将其添加到清单 6.9 的程序中。

清单 6.10 加这段代码到清单 6.9 中使其能够传输数据到服务器

```

public int dataSocketOut(String datCmd,Reader r) throws IOException {
    int status;
    status=dataSocket(datCmd);
    if (status!=0) return status;

```



```

datWriter=new OutputStreamWriter(datskt.getOutputStream());
int ch;
do {
    ch=r.read();
    if (ch!=-1) datWriter.write((char)ch);
} while (ch!=-1);
datWriter.flush();
datskt.close();
return getResponse();
}

```

要看程序的实际效果，可以使用清单 6.11 中的 main 主函数部分替换清单 6.9 中的 main 主函数。它会从名为 ftp.test 的服务器那里读一个文件，并将读出的文件写回到名为 ftp1.test 的服务器那里。注意，因为 dataSocketIn 和 dataSocketOut 都接受流操作，向控制台发送一个文件或者接收一个目录列表都比较容易。创建 StringReader 或者 StringWriter 对象，从而使用串，以代替文件作为数据源或者数据的终点进行操作，同样很容易。

清单 6.11 在清单 6.9 和 6.10 中使用下面的主函数来发送、接收文件

```

public static void main(String[]args)throws Exception{
    if(args.length<3||args.length>4){
        System.out.println("usage:EZftp host user password[directory]");
        System.exit(1);
    }
    FtpLS obj=new FtpLS();
    Obj.passive=true;
    If(!obj.connect(args[0],args[1],args[2])){
        System.out.println("Can't connect");
        System.exit(1);
    }

    FileWriter f=new FileWriter('ftp.test');
    Obj.dataSocketIn("RETR readmc",f);
    System.out.println(obj.lastResponse);

    //Write test
    FileReader fw=new FileReader("ftp.test");
    Obj.dataSocketOut("STOR ftp1.test",fw);
    System.out.println(obj.lastResponse);

    System.exit(0);
}

```

6.2.7 选择主动方式还是被动方式

你如果检查一下清单 6.9 中的代码，会注意到例程 dataSocket 实现了实际上的数据连接。这个函数的操作与依赖于 passive 标志的状态不一致。

使用主动模式时采取的步骤（此时 passive 值为 false）：

1. 创建一个 `ServerSocket`，参数为 0 导致对象选择一个可得到的端口。
2. 确定套接字使用的端口（调用 `getLocalPort`）。
3. 通过调用控制套接字上的 `getLocalAddress` 方法的来确定用于和服务器通信的 IP 地址，可以通过调用 `getHostAddress` 进一步将其简化成一个简单的 IP 地址。
4. 创建一个 `PORT` 命令串，其参数（由逗号分隔）是 IP 地址的 4 个 8 位值、16 位端口号的高 8 位（来自步骤 2）和 16 位端口号的低 8 位。发送这个命令并等待一个肯定的响应（响应码值为 200）。

5. 调用 `accept` 来等待数据连接。如果服务器永远不打开连接，就可能需要提供超时处理。在被动模式下，采取如下步骤：

1. 发送 `PASV` 命令，等待肯定响应。
2. 响应将包含使用格式 `PORT` 命令相同的 IP 地址和端口号，将数据从响应中解析。
3. 使用第二步得到的 IP 地址和端口号，创建一个常规 `Socket` 的实例。

使用 `PORT` 命令在防火墙下可能不会工作，因此，如果 `PORT` 命令失败，尝试使用被动模式是一个好方法。当然，有些 FTP 服务器可能没有实现被动模式（虽然很少见），因此如果有必要，可以准备求助于使用 `PORT` 命令（在被动模式没有实现的情况下）。

6.2.8 使用 FTP 的开放源码

有若干 FTP 的开放源码包可以得到。有一个看起来功能很强大，位置在 `Giant Java Tree`（www.git.org）。这个包将 FTP 的逻辑很好地封装了起来。

要使用包 `org.gjt.tst.net.ftp.client` 简单地将类 `FtpClientExtraProtocol` 实例化。这允许你使用诸如 `user`、`password`、`download` 和 `upload` 此类方法。它也提供了一个枚举，用于处理目录列表，如果你不需要类 `FtpClientExtraProtocol` 中的那些附加功能（主要是目录操作函数），你也可以使用类 `FtpClientProtocol`。

清单 6.12 显示了使用 `FtpClientExtraProtocol` 来下载一个文件的简单程序。该方法与用户命令相对应，因此使用这个类工作比较容易。也可以使用这个包来创建 FTP 服务器，尽管所做工作要稍微多一些。

清单 6.12 本程序使用开放源码的 FTP 类下载一个文件

```
import java.io.PrintWriter;
import java.io.OutputStreamWriter;
import java.io.FileOutputStream;
import java.io.File;
import java.io.IOException;
import org.gjt.tst.net.ftp.client.*;

public class FtpClient {
    private FtpClientExtraProtocol ftp;
    // This does the actual download
    private static void downloadTheFile()
        throws IOException, InterruptedException {
```

```
File file = new File("test.txt");
FileOutputStream out = null;
try {
    out = new FileOutputStream(file);
    ftp.download(out, "test.txt");
}
finally {
    if (null != out) {
        out.close();
        out = null;
    }
}

}

public static void main(String[] args) {

    PrintWriter trace =
        new PrintWriter(new OutputStreamWriter(System.err), true);

    try {
        ftp = new FtpClientExtraProtocol("localhost", trace);
        ftp.setTimeout(2000);
    }
    catch (IOException e) {
        trace.println("Cannot connect to server");
        return;
    }

    ftp.setDataConnectionActiveServer();
    try {
        ftp.user("kirk");
        if (!ftp.isLoggedIn())
            ftp.password("enterprise");
        downloadTestFile(ftp);
    }
    catch (InterruptedException ex) {
        trace.println(ex.getMessage());
    }
    catch (IOException ex) {
        trace.println(ex.getMessage());
    }
    finally {
        try {
            ftp.logout();
            ftp.close();
        } catch (IOException e) {}
    }
}
}
```

第 7 章 SMTP 协议

7.1 深入介绍

我妻子说我被邮件迷住了，为何不呢？邮件意味着有人想告诉你某些事情（不包括垃圾邮件），它比较重要，足以放入信件中。当我还是个小孩子的时候，我习惯于寄出免费的东西，并且每天焦急地等待邮递员（我们那时没有视频游戏）。

有些人认为邮件已经过时了，有什么事情都可以打电话。但我想电话是引起很多社会问题的一个因素。当你打电话给某人，收到的是忙信号，那是被拒绝的一种微妙的形式。当然，呼叫等待将拒绝从第二个呼叫者转移到第一个呼叫者，现在你可选择你想侮辱的任何一个呼叫者。

幸运的是，邮件为信件书写的艺术带来了新的契机。人们不想再等待邮递员了；他们等待着接收邮件的通知，那很可能是为什么 AOL 如此流行的原因。“你已经得到邮件了”能立即让你满意，让你非常微妙地感到你是一个主人翁。

当你发送邮件的时候，要连接到 SMTP（简单邮件传输协议）主机的 25 号端口上。这个主机将邮件传递给本地用户，如果接收者是另一台计算机上的用户，那就中转它。可以在 Internet 的 RFC2821 上找到所有关于 SMTP 的技术细节，上边对 RFC 的更新日期可以回溯到 1982 年。看了这个协议，你会知道作者想让一些人使用电传打字机拨进 SMTP 主机（记得这是 1982 年），使用软件手工管理交易。也可以使用 SMTP 发送一定数量的早期形式的立即消息。今天你看不到这些用法，因为很多立即消息是采用 P2P（peer-to-peer）的解决方案。

如果 SMTP 的全部目标是将一条消息从发送者发到接收者，可能使用 FTP 协议会更好。但是，作者也允许一个 SMTP 主机将邮件中转给另一个主机。因此，协议不仅要负责手工的登录，还要负责机器说明。你如果使用 Telnet 程序登录到一台 SMTP 主机（端口 25），将看到如下的一个响应：

```
220 smtp1b.mail.yahoo.com ESMTP
```

这个响应来自 Yahoo，SMTP 服务器的地址为 `smtp.mail.yahoo.com`。

主机发送的每一行都有三位的响应码，如果这个响应有多行，那么第四个字符就是连字符。如果第四个字符是空格字符，那么该行是多行响应的最后一行或者是唯一一行响应。程序通常只读开头四个字符，忽略该行的剩余部分。可能如你所料，该行剩余部分对人来讲更有用。在表 7.3（快速解决方案那一节里）里可以找到响应码及其意义的清单。这些码值与 FTP 的码值很相似（见第 6 章），如你所料，它们遵从相同的模式。以 2 开始的码值表示成功，以

3 开始的码值要求有更多的信息，而 4XX 和 5XX 两组值表示失败（5XX 表示严重的失败）。

像 FTP 里一样，第二位响应码也有特殊的意义：为 0 表示发生了语法错误，为 1 表示信息消息，为 2 表示相关的连接，最后，第二位为 5 表示邮件系统的状态。

你可以向 SMTP 服务器发送很多种可能的命令（见表 7.1）。但是，很多命令并不常用，命令次序很重要，基本次序如下：

1. HELO 识别发送方的机器。
2. MAIL FROM: 从发送方开始邮件。
3. RCPT TO: 识别接收方（可以多次发生）。
4. DATA 指定邮件消息。
5. QUIT 终止与服务器的会话。

表 7.1 SMTP 命令比较少，大多数程序需要的命令就更少

命令	说明	需要吗？
DATA	开始有邮件消息的实际内容（最后一行只包含一个点号表示结束）	Y
EXPN	扩展邮件列表	
EHLO	识别发送者并触发扩展的 SMTP 选项	Y
HELO	通过主机名或者 IP 地址识别发送者	Y
HELP	发送一个帮助响应	
MAIL FROM:	从发送方开始一个邮件事务	Y
NOOP	产生一个肯定响应，但不采取行动	Y
QUIT	终止会话	Y
RCPT TO:	指出邮件的单一接收者	Y
RSET	放弃当前事务	Y
SEND	直接发送数据到终端	
SOML	发送数据到终端或者通过邮件	
SAML	发送数据到终端并且通过邮件	
TURN	让发送者成为接收者并且反之亦然	
VERFY	检验邮件地址	

每个命令只有一或两个不是出错的响应。大多数邮件程序让你指定多个接收者，也有抄送和暗送（CC 和 BCC）的多个接收者。就 SMTP 来看，你指定的所有这些接收者都是使用单独的 RCPT TO: 命令。

CC 接收者（抄送）和 BCC 接收者（暗送）的不同在于接收者的地址是否出现在邮件的

题头里，像一个 Web 文档一样，邮件有自己的题头和正文（可以在 RFC2821 中找到相关细节）。你可以发送很多题头，包括主题、响应路径以及其他题头（不同的邮件程序可能解释的方式不一样）。SMTP 并不在意你要包括哪些题头（如果有题头）。

7.1.1 验证

除了发送邮件，SMTP 主机可以使用 VRFY 命令来验证邮件地址。该命令会让主机查询用户，并提供一个完全的地址。但有些站点并不允许这样做，因为这会简化黑客们轮询用户的 ID。

例如，这里是对验证站点 smtp.mail.yahoo.com 中的一个地址的尝试：

```
220 smtp016.mail.yahoo.com ESMTP
VRFY jtkirk
252 send some mail, i'll try my best
```

它并不是一个很有帮助的消息。

SMTP 有几个特征并不很常见。一个是 EXPN 命令，它允许扩展一个邮件列表（通常产生多行响应），另一个特征是 SMTP 地址可以带有路径，你可以指定 SMTP 服务器在该路径上发送邮件到另一台机器，那台机器再将邮件发送到另一台机器，依此类推。当计算机连接质量不好时，这就显得很有用，但现在几乎所有计算机都以某些方式连到公用 Internet 网上，很少会出现这种情形。

7.1.2 超时、多行和透明性

当我看到 Web 上其他 SMTP 例子时，注意到它们有很多没有遵从 RFC 关于邮件的规定。它们在大多数情况下能够工作，但一些奇怪的条件可能会导致失败。我在自己的代码里试图避免这些情况。特别地，我注意到很多例子没能正确的处理下列情形：

- **Time-out（超时）**——不管有多少原因，服务器响应可能失败。它谨慎地允许服务器终止连接前有一定的响应时间，有些程序不允许有这个可能，其他一些程序允许超时但只是使用简单循环，效率不高。
- **Multiple Lines（多行）**——像我早些时候提到的一样，有些服务器对单个响应作出多行答复，每个响应开始都是 3 位的响应码，如果不是最后一行，第 4 个字符将是连字符，有些 SMTP 代码的例子没有考虑这个可能性。
- **Transparency（透明性）**——当处理邮件消息时，SMTP 服务会查找只带有一个点号的行来终止消息。但是，如果有一行消息，它实际上就是一个点号消息呢？为避免这个问题，通常的 SMTP 客户端在以点号开始的任何一行都加上一个附加的点号。如果服务器检查一行并发现开头有两个点号，它知道省略第一个点号（因为它是由客户端程序很安全的添加上去的），接着处理余下更多的行。

7.1.3 扩展的 SMTP

SMTP 的更新版本添加了新特征，同时又不牺牲向后兼容性。如果客户端通过 EHLO 而不是 HELO 来识别客户端，服务器会认为扩展的 SMTP 协议也有效。

当发出 EHLO 命令后，可以等待多行的响应。服务器将对它能了解的所有扩展符作出响应（例如，8BITMIME 表示服务器能接受 8 位的附件）。

7.1.4 题头

SMTP 并不太注意你发送的邮件消息的正文里的内容。但很多邮件程序都希望发送带有特定意义的邮件题头（标题）。在 RFC2822 里可以找到一个完整的清单。下面是一些很常见的邮件题头：

- Subject——消息的主题。
- From——发送者的邮件地址。
- To——消息的主要接收者。
- CC——抄送方式的接收者。这些接收者将出现在地址列表里，接收到消息的一个拷贝。
- BCC——暗送方式的接收者。这些接收者将收到一个拷贝，但不会出现在邮件的列表里。
- Return-Path——应该接收管理消息的邮件地址。
- Received——接收的日期和时间。
- Reply-To——接收任何响应的邮件地址。
- Date 消息发送的日期和时间。

空行将题头和消息正文分开，下面是一个典型的含有最少题头的邮件消息：

```
Date: 26 Jul 99 1429 EDT
From: alw@al-williams.com
To: admin@al-williams.com
```

```
The server is down again -D Al
```

题头是发送格式化邮件和附件的关键。考虑这个邮件消息（包括题头）：

```
Subject: html test
mime-version: 1.0
content-type: text/html
content-transfer-encoding: 7bit
```

```
<H1>Hello Al</H1>
<hr>
<P>This is not spam.... it's wham!</P>
```

这则消息会作为格式化网页出现在大多数邮件程序中。你可以在 RFC2045 至 RFC2049 里找到这些题头的有关细节（所谓的 MIME）。

因为有些邮件网关只接收 7 位的 ASCII 文本，任何二进制数据，如一个图像附件，不得不使用某种方案将 8 位字节映射成 7 位字符。

7.1.5 编码

因为有些机器以及通信处理邮件的命令行可能不允许使用 8 位数据，RFC1521 里定义的几个方案允许将 8 位字节转换成 7 位字符。可能如你所想，做这种编码会让数据的大小膨胀。如果你仅仅将每个字节转换成两个 16 进制位，可能会使数据变成双倍大小，那样效率并不高，因此每种方案都尽量将使影响最小化。

7.1.5.1 可引用可打印 (Quotable Printable) 编码

这个简单的方案效率并不太高，除非 8 位字符很少出现在邮件消息里。这个方法通常并不将字符替换成 21 到 3C 以及 3E 到 7E (可打印字符) 之间的 16 进制值，但是，任何字节都可以替换成一个等号和两个字符的 16 进制数。因此，尽管你不用将 A (16 进制为 41) 以这种方式编码，你可以将它写成=41。一个 BEL (响铃) 字符将是=07，必须被编码。

如果那些字符不是在行末，也可以忽略空字符 (空格以及制表符)。有些系统可能在行末加上伪空格，这样接收者会在行末去掉通常的空格字符。如果需要一个硬行中断，必须将它作为回车符和换行符进行编码，不管你本地系统是否将它看作是行末。另外，任何行不能超过 76 个字符 (不包括回车符和换行符，但要包括任何 ESC 字符)。如果有更长的行，可以将它分割成多行，并对附加的行使用等号结束。这就是所谓的软行中断，因为接收者可以选择忽略它。

如果使用这种方式编码，那就要将内容传输编码题头设置成可引用可打印。

7.1.5.2 Base 64 编码

可引用可打印编码对大部分都是纯文本的消息比较有用，但它可以很快将一个二进制文件变得很大 (最糟的情况下，文件大小可能增至 3 倍)。Base 64 编码实现起来更复杂，但它只会将文件大小扩大 33%。

Base 64 编码使用 ASCII 字符集中的 65 个字符。这意味着每个字符可以代表 6 位 (等号这个字符不代表数据)。这个算法将 3 个字节合在一起，形成一个 24 位的字。接着它将 24 位的字拆成 4 个 6 位的标志 (总共还是 24 位)。每个标志都由一个 ASCII 字符来表示 (见表 7.2)。RFC 要求第一位是第一个字中最重要的位，因此不会余下什么内容。

为了适应邮件系统，输出行不会多于 76 个字符。行中断不计，因此可以将行切分得比较恰当，在文件的末尾，可能没有足够的字节组成一个三字节组。在这种情况下，可以在末尾添加 0，直到末尾出现一个完整数目的 6 位组。如果有一个附加的字符，那就要加上 4 个二进制 0，并执行编码，得到两个 base 64 字符，为保持编码组都是 4 个字符，你可以在输出后边补上两个等号。如果最后有两个附加的字符，那就加上两个 0，编码，并补上一个等号。当然，如果你的文件大小刚好能被 3 整除，就不用在后边填补字符了。

表 7.2 你可以使用这张表为 base 64 编码算法将 6 位组编码成 ASCII 字符

16 进制值 编码	16 进制值 编码	16 进制值 编码	16 进制值 编码
00 A	11 R	22 i	33 z
01 B	12 S	23 j	34 0
02 C	13 T	24 k	35 l
03 D	14 U	25 l	36 2
04 E	15 V	26 m	37 3
05 F	16 W	27 n	38 4
07 H	18 Y	29 p	3A 6
08 I	19 Z	2A q	3B 7
09 J	1A a	2B r	3C 8
0A K	1B b	2C s	3D 9
0B L	1C c	2D t	3E +
0C M	1D d	2E u	3F /
0D N	1E e	2F v	
0E O	1F f	30 w	
0F P	20 g	31 x	
10 Q	21 h	32 y	

在本章的快速解决方案一节里可以见到执行这种编码的类。

7.1.6 实现

在处理 SMTP 类之前，我写了两个较小的帮助类。第一个类是 `MailMessage`（见清单 7.1），简单地将串封装，以组成典型的邮件消息；第二类是 `SMTPResults`（见清单 7.2），它是一个静态的整数容器，代表着表 7.3 中的响应码。

主类 `SMTP`（见表 7.3），只有两个比较重要的方法。一个是构造函数，用于创建 `SMTP` 实例的构造函数要求有 `SMTP` 服务器的主机名或者 IP 地址作为参数，另一个方法是 `sendMail`，只要将装填好的 `MailMessage` 对象传递给 `sendMail` 方法，它完成剩余的工作，如果 `sendMail` 方法返回值为 0，表明工作正常，如果有错，可以在返回值里找到相应的 `SMTP` 响应码。

`SMTP` 的内部细节更有意义，`sendMail` 函数简单地打开一个套接字，接着调用方法 `sendMailEngine`。即使 `sendMailEngine` 返回一个错误，函数 `sendMail` 也能恰当地关闭套接字（你也可以使用 `finally` 语句做相同的事）。方法 `sendAddress` 将邮件地址列表拆开，并使用命令 `RCPT TO:` 分开发送消息。

可能最有趣的例程是 `getResponse`。这个函数从 SMTP 服务器里读一个完整的响应，并将它与期望的响应码进行检验，如果它与期望的响应码中的一个相匹配，函数返回值为 0，否则就返回该响应码。

与这个函数有关的一个问题是超时处理：函数在放弃之前必须等待一段时间，处理这个问题的一种方法是检验套接字中的字符，直到一段确定时间过后。但是，这会让你的程序在等待的过程中持续运行，消耗系统资源。我决定编写一个小的私有类 `TimeoutRead`，对类 `Thread` 进行扩展。这个对象在一个独立的线程里执行，它在套接字里调用 `readLine`。一旦代码发现响应结束，线程也就结束了。

接着，主程序必须等待线程结束或者一段时间的超时，这就是函数 `Thread.join` 所做的工作，主程序调用 `join`，传递一个要等待的毫秒数作它的参数，当 `join` 返回时，要么线程结束，要么超时时间段过期。

这个设计的唯一问题是线程并不只是因为超时时间过期而结束，尽管 `Thread` 对象有 `stop` 和 `suspend` 方法，但一般都不太赞成使用，另外，也不能保证这两个方法会以某种方式中断 `readLine`。如果线程还在运行，它可能干扰程序的关闭（更不用说它浪费系统资源了）。即使调用 `System.exit` 来中止线程，也不是对所有的程序都适用。

这就是为何 `sendMail` 在所有条件下关闭套接字比较重要的原因，当套接字关闭的时候，函数 `readLine` 抛出一个异常，并终止该线程。

类 `SMTP` 也有一个主函数来测试该类，它以邮件地址来作为命令行参数，并发送一个短测试消息到那个邮件地址，当然，如果你使用该类作为更大系统的一部分时，`main` 函数不会被调用，你可以安全地将其删除。

类 `MailMessage` 有两个成员用于一些高级特征，方法 `addHeader` 允许你向消息加入附加的题头，如果你要发送如 HTML 之类的消息，这将很有用，另一个高级特征是对一个 `Writer` 类对象的引用，对消息正文进行编码（使用 base 64 编码，举个例子）。如果将这个引用缺省设为 `null`，该类就不对消息编码，如果提供了一个对该类的引用（例如，`B64Encode.class`），那你必须也使用 `addHeader` 添加必需的题头。

清单 7.1 类 `MailMessage` 包含了 SMTP 类用于发送邮件的信息

```
import java.util.Hashtable;
import java.io.Writer;

/**
 * This is just a collection of data that makes up a mail message.
 * @author Al Williams
 * @version 1.0
 * @see SMTP#sendMail
 */
public class MailMessage
{
```



```
/**
 * Default constructor.
 * @param None
 */
public MailMessage()
{
}
/**
 * Constructor that initializes.
 * @param _from The sender's e-mail address.
 * @param _to The recipient's e-mail address (separate
 *      multiple addresses with semicolons).
 * @param _cc Carbon copy addresses (separate multiple addresses
 *      with semicolons).
 * @param _bcc Blind copy addresses (separate multiple addresses
 *      with semicolons).
 * @param _subject The email subject.
 * @param _body The body of the email.
 */
public MailMessage( String _from, String _to,
String _cc, String _bcc, String _subject, String _body )
{
    sender = _from;
    to = _to;
    cc = _cc;
    bcc = _bcc;
    subject = _subject;
    body = _body;
}
/**
 * Constructor with commonly required arguments.
 * @param _from The sender's e-mail address.
 * @param _to The recipient's e-mail address
 *      (separate multiple addresses with semicolons).
 * @param _subject The e-mail subject.
 * @param _body The body of the e-mail.
 */
public MailMessage( String _from, String _to, String _subject,
String _body)
{
    sender = _from;
    to = _to;
    subject = _subject;
    body = _body;
}
/**
 * The sender's e-mail address.
 */
```

```
public String sender;
/**
 * The recipient's e-mail address (separate multiple addresses
 * with semicolons).
 */
public String to;
/**
 * The Carbon Copy addresses (separate multiple addresses with
 * semicolons).
 */
public String cc;
/**
 * The Blind Carbon Copy addresses (separate multiple addresses
 * with semicolons).
 */
public String bcc;
/**
 * The e-mail's subject.
 */
public String subject;
/**
 * The e-mail's body.
 */
public String body;
/**
 * Extra headers
 */
public Class encoder=null;
public Hashtable headers=null;
public void addHeader(String header,String value) {
    if (headers==null) headers=new Hashtable();
    headers.put(header,value);
}
}
```

清单 7.2 这个类提供了 SMTP 结果编码

```
/**
 * Static values for SMTP result codes
 * @author Al Williams
 * @version 1.0
 */
public class SMTPResults
{
    /**
     * Syntax error, command unrecognized
     */
    static final public int SMTP_RESULT_UNRECOG = 500;
}
```

```
* Syntax error in parameters or arguments
*/
static final public int SMTP_RESULT_PARAM = 501;
/**
 * Command not implemented
 */
static final public int SMTP_RESULT_UNIMPLEMENTED = 502;
/**
 * Bad sequence of commands
 */
static final public int SMTP_RESULT_SEQUENCE = 503;
/**
 * Command parameter not implemented
 */
static final public int SMTP_RESULT_PARAMNI = 504;
/**
 * System status, or system help reply
 */
static final public int SMTP_RESULT_SYSTEM = 211;
/**
 * Help message
 */
static final public int SMTP_RESULT_HELP = 214;
/**
 * <domain> Service ready
 */
static final public int SMTP_RESULT_READY = 220;
/**
 * <domain> Service closing transmission channel
 */
static final public int SMTP_RESULT_CLOSING = 221;
/** <
 * <domain> Service not available, closing transmission channel
 */
static final public int SMTP_RESULT_SERUNAVAILABLE = 421;
/**
 * Requested mail action okay, completed
 */
static final public int SMTP_RESULT_COMPLETED = 250;
/**
 * User not local; will forward to <forward-path>
 */
static final public int SMTP_RESULT_FORWARD = 251;
/**
 * Requested mail action not taken: mailbox unavailable
 */
static final public int SMTP_RESULT_MBXUNAVAILABLE = 450;
/**
```

```

    * Requested action not taken: mailbox unavailable
    */
    static final public int SMTP_RESULT_NOTTAKEN = 550;
    /**
    * Requested action aborted: error in processing
    */
    static final public int SMTP_RESULT_ABORTED = 451;
    /**
    * User not local; please try <forward-path>
    */
    static final public int SMTP_RESULT_USER_NOT_LOCAL = 551;
    /**
    * Requested action not taken: insufficient system storage
    */
    static final public int SMTP_RESULT_STORAGE = 452;
    /**
    * Requested mail action aborted: exceeded storage allocation
    */
    static final public int SMTP_RESULT_EXSTORAGE = 552;
    /**
    * Requested action not taken: mailbox name not allowed
    */
    static final public int SMTP_RESULT_NOT_ALLOWED = 553;
    /**
    * Start mail input; end with <CRLF>.<CRLF>
    */
    static final public int SMTP_RESULT_MAIL_START = 354;
    /**
    * Transaction failed
    */
    static final public int SMTP_RESULT_TRANS_FAILED = 554;
}

```

清单 7.3 类 SMTP 使得用一个 Java 程序发送邮件很容易

```

import java.awt.*;
import java.net.*;
import java.util.*;
import java.io.*;
import SMTPResults;
import MailMessage;

/**
 * SMTP Class. You can use this class to
 * send e-mail via a SMTP server.
 * You can find more info about Email and SMTP via the RFCs particularly
 * <A HREF=http://www.faqs.org/rfcs/rfc821.html>RFC821</A> and
 * <A HREF=http://www.faqs.org/rfcs/rfc822.html>RFC822</A>
 * @author Al Williams

```

```

* @version 1.0
*/
public class SMTP
{
    // things you might want to change
    // 30 seconds timeout
    final static boolean debug = true;
    final static int WAIT_TIMEOUT = ( 30 * 1000 );
    final static int smtpPort = 25;
    final static String addressSep = ";"; // separates e-mail addresses
    // SMTP server for testing
    final static String testServer = "mail.direcpc.com";

    String smtpServer;
    // could hardcode this
    MailMessage message;
    String hostname;
    BufferedReader input;
    OutputStream output;
    String errorText;    // copy of last response -- in case of error
    Socket sock;

    final String crlf = "\r\n";
    /**
     * Get the last response message. Useful for displaying error
     * from SMTP server.
     * @returns String containing last response from server.
     */
    public String getLastResponse()
    {
        if (errorText==null) return "Unable to connect or unknown error";
        return errorText;
    }

    /**
     * Constructor. Requires SMTP server name.
     * @param host An SMTP server. Remember, the server must be accessible
     * from this code.
     * In particular, applets can usually only connect back
     * to the same host they originated from.
     */
    public SMTP( String host )
    {
        smtpServer = host;
    }
}

```



```
//read data from input stream to buffer String

private int getResponse( int expect1 ) throws IOException
{
    return getResponse( expect1, -1 );
}

synchronized private int getResponse( int expect1, int expect2 )
    throws IOException
{
    boolean defStatus;
    long startTime;
    int replyCode;

    TimeoutRead thread = new TimeoutRead( input );
    thread.setBuffer("");
    thread.start();
    try
    {
        thread.join( WAIT_TIMEOUT );
    }
    catch( InterruptedException e )
    {
    }
    if( thread.isComplete() && thread.getBuffer().length() > 0 )
    {
        try
        {
            // if there is an error, this is it.
            errorText = thread.getBuffer();
            replyCode = Integer.valueOf(
                errorText.substring( 0, 3 ) ).intValue();
            if( replyCode == 0 ) return -1;
            if( replyCode == expect1 || replyCode == expect2 )
                return 0;
            return replyCode;
        }
        catch( NumberFormatException e )
        {
            error(e);
            return -1;
        }
    }
    return -1;
    // nothing in buffer
}
```

```

private void writeString( String s ) throws IOException
{
    output.write( s.getBytes() );
}
private int sendAddresses(String pfx, String addr) throws IOException
{
    int n0=0;
    int n;
    int rv=0;
    while (rv==0 && (n=addr.indexOf(addressSep,n0))!=-1)
    {
        writeString(pfx+addr.substring(n0,n)+crlf);
        n0=n+1;
        rv=getResponse( SMTPResults.SMTP_RESULT_COMPLETED,
            SMTPResults.SMTP_RESULT_FORWARD );
    }
    if (rv==0)
    {
        writeString(pfx+addr.substring(n0)+crlf);
        rv=getResponse( SMTPResults.SMTP_RESULT_COMPLETED,
            SMTPResults.SMTP_RESULT_FORWARD );
    }
    return rv;
}

/**
 * Use sendMail to actually send an e-mail message.
 * @param MailMessage This is a filled-in MailMessage object that
 *     specifies the text, subject, and recipients.
 * @return Zero if successful. Otherwise, it returns the SMTP
 *     return code.
 * @see SMTPResults
 */

public int sendMail( MailMessage msg)
{
    String inBuffer;
    String outBuffer;
    int rv;
    message = msg;
    if( msg.to == null || msg.to.length() == 0 )
    {
        error("Must supply To field");
        return -1;
    }
    // Create connection
    try
    {

```

```
        sock = new Socket( smtpServer, smtpPort );
        hostname = "[" + sock.getLocalAddress().getHostAddress() + "]";
    }
    catch( IOException e )
    {
        error(e);
        return -1;
    }
    //Create I/O streams
    try
    {
        input = new BufferedReader( new InputStreamReader(
            sock.getInputStream() ) );
    }
    catch( IOException e )
    {
        error(e);
        return -1;
    }
    try
    {
        output = sock.getOutputStream();
    }
    catch( IOException e )
    {
        error(e);
        return -1;
    }
    rv=sendMailEngine();
// end connection
    try
    {
        sock.close();
        sock=null;
    }
    catch( IOException e )
    {
        error(e);
        return -1;
    }
    return rv;
}

// this is a separate routine so the main sendMail can always close the socket
private int sendMailEngine()
{
    try
```

```

{

    int replyCode;
    int n;
    Date today = new Date();
    replyCode = getResponse( SMTPResults.SMTP_RESULT_READY );
    if( replyCode != 0 ) return replyCode;
    //Send HELO
    writeString( "HELO " + hostname + crlf );
    replyCode = getResponse( SMTPResults.SMTP_RESULT_COMPLETED );
    if( replyCode != 0 ) return replyCode;
    // Identify sender
    writeString( "MAIL FROM: " + message.sender + crlf );
    replyCode = getResponse( SMTPResults.SMTP_RESULT_COMPLETED );
    if( replyCode != 0 ) return replyCode;
    // Send to all recipients
    replyCode = sendAddresses("RCPT TO: ",message.to);
    if( replyCode != 0 ) return replyCode;
    // Send to all CC's (if any)
    if( message.cc != null && message.cc.length() != 0 )
    {
        replyCode = sendAddresses("RCPT TO: ",message.cc);
        if( replyCode != 0 ) return replyCode;
    }
    // Send to all BCC's (if any)
    if( message.bcc != null && message.bcc.length() != 0 )
    {
        replyCode = sendAddresses("RCPT TO: ",message.bcc);
        if( replyCode != 0 ) return replyCode;
    }

    // Send message
    writeString( "DATA" + crlf );
    replyCode = getResponse( SMTPResults.SMTP_RESULT_MAIL_START );
    if( replyCode != 0 ) return replyCode;

    //Send mail content CRLF.CRLF
    // Start with headers
    writeString( "Subject: " + message.subject + crlf);
    writeString( "From: " + message.sender + crlf);
    writeString( "To: " + message.to + crlf);
    if( message.cc != null && message.cc.length() != 0 )
        writeString( "Cc: " + message.cc + crlf);
    writeString( "X-Mailer: SMTP Java Class by Al Williams" +
        crlf);
    writeString( "Comment: Unauthenticated sender" + crlf );
    if (message.headers!=null) {
        Enumeration key=message.headers.keys();

```

```

        while (key.hasMoreElements()) {
            String keystring=(String)key.nextElement();

writeString(keystring+": "+(String)message.headers.get(keystring)+crlf);
        }
    }

    writeString( "Date: " + today.toString() +crlf + crlf );
String bodybuf=message.body;
if (message.encoder!=null) {
    StringWriter wbuf = new StringWriter();
    Writer xform;
    try {
        Class[] args = new Class[1];
        args[0]=Writer.class;
        java.lang.reflect.Constructor c=
            message.encoder.getConstructor(args);
        Object[] cargs=new Object[1];
        cargs[0]=wbuf;
        xform=(Writer)c.newInstance(cargs);
        xform.write(message.body);
        xform.flush();
    }
    catch (Exception e)
        error(e);
        return -1;
    }
    bodybuf=wbuf.toString();
}

    StringBuffer body=new StringBuffer(bodybuf);
// chop lines to 76 characters
int sbindex=0;
int lineindex;
do {
    String bodyline=body.substring(sbindex);
    String line;
    lineindex=bodyline.indexOf('\n');
    if (lineindex== -1)
        line=bodyline;
    else
        line=bodyline.substring(0,lineindex);
    while (line.length()>76) {
        if (line.charAt(0)=='.') writeString(".");
        writeString(line.substring(0,76));
        line=line.substring(76);
    }
    if (line.charAt(0)=='.') writeString(".");
}

```



```

        writeString(line); // write rest
        sbindex+=lineindex+1;
        writeString(crlf);
    } while (lineindex!=-1);
        writeString( "." + crlf ); // end mail
        replyCode = getResponse(
            SMTPResults.SMTP_RESULT_COMPLETED);
        if( replyCode != 0 ) return replyCode;
        // Quit
        writeString( "QUIT" + crlf );
        replyCode = getResponse( SMTPResults.SMTP_RESULT_CLOSING );
        if( replyCode != 0 ) return replyCode;
    }
    catch( IOException e )
    {
        error(e);
        return -1;
    }
    return 0;
}

protected void finalize() throws Throwable
{
    if (sock!=null) sock.close();
    super.finalize();
}

private void error(Exception e) {
    if (debug) error(e.getMessage());
}
private void error(String s) {
    if (debug) {
        System.out.println(s);
    }
}

public static void main( String args [] )
{
    int rc;
    Date today = new Date();
    if (args.length!=1)
    {
        System.out.println("Usage: SMTP e-mail address");
        System.exit(1);
    }
    System.out.println("Sending test message to " + args[0]);
    MailMessage msg = new MailMessage( "alw@al-williams.com", args[0],
    "Test",

```

```

        '<h1>Sent at ' + today.toString()+"</h1><hr>Thanks!");
msg.addHeader("MIME-Version","1.0");
msg.addHeader("Content-Type","text/html");
msg.addHeader("Content-Transfer-Encoding","Base64");
msg.encoder=B64Encoder.class;
    SMTP smtp = new SMTP( testServer);
    rc = smtp.sendMail( msg );
    if( rc != 0 )
    {
        System.out.println( "Error " + rc );
        System.out.println(smtp.getLastResponse());
    }
    else
        System.out.println( "OK" );
}

}

// private class to handle the reading in a thread
class TimeoutRead extends Thread
{
    private String buffer = new String( "" );
    private BufferedReader input;
    private boolean complete = false;
    synchronized String getBuffer() { return buffer; }
    synchronized void setBuffer(String s) { buffer=s; }
    synchronized public boolean isComplete()
    {
        return complete;
    }
    public TimeoutRead( BufferedReader i )
    {
        input = i;
    }
    public void run() // do input in thread
    {
        try
        {
            do
            {
                setBuffer(input.readLine());
            } while (getBuffer().charAt(3)=='-'); // loop on multi-line response
        }
        // This line is useful for debugging
        if (SMTP.debug) System.out.println(buffer);
    }
    catch( IOException e )
    {

```

```

        setBuffer("");
    }
    complete = true;
}

}

```

7.1.7 使用 SMTP

在清单 7.4 里会看到一个 JSP 页面使用 SMTP 组件，它从一个属性文件里读一组名字列表，用户填写邮件域的值并从一个下拉框里选择一组名字，JSP 文件将邮件发送给列表里的每个人，如果要改变列表的成员，就只能改变属性文件了，只需一点附加工作你就能写一个基于 Web 的接口，可以使用 JSP 来浏览和更新属性文件。

当然，SMTP 的类文件以及它所依赖的所有类文件，它们所在的目录必须出现在服务器的 CLASSPATH 中，对很多服务器而言，那将意味着在应用的主目录中将它放入 WEB-INF/classes 目录里边，可以用下面几行完成一个简单的 JSP 页面来提示类的路径：

```

<%
out.println(System.getProperty ("java.class.path"));
%>

```

清单 7.4 你可以在一个 JSP 页里使用 SMTP 来发送邮件

```

<%@ page import="java.io.InputStream" %>
<%@ page import="java.util.Properties" %>
<%@ page import="java.util.Enumeration" %>

<%!
    Properties database;
%>
<%
    ServletContext ctx;
    InputStream is;
    String dfile;
    dfile="/maillist/addresses.prop";
    database=new Properties();
    ctx=getServletContext();
    is=ctx.getResourceAsStream(dfile);
    if (is==null)
        out.println("Internal server error " + dfile);
    else
    {
        database.load(is);
        is.close();
    }
%>

```

```

<HTML>
<HEAD>
</HEAD>
<BODY BgColor=CornSilk>
<% if (request.getContentLength() == -1) { %>
<H1>Send mail to a group</H1>
<FORM METHOD=POST>

Group: <SELECT NAME=group>
<%
    Enumeration e=database.keys();
    while (e.hasMoreElements())
    {
        String s=(String)e.nextElement();
        out.println('<OPTION VALUE="' + s + ">" + s);
    }
%>

</SELECT>
<BR>
From: <INPUT NAME=From SIZE=40><BR>
Subject: <INPUT NAME=Subj SIZE=40><BR>
<TEXTAREA ROWS=25 COLS=80 Name=msg>
</TEXTAREA>
<BR>
<INPUT TYPE=SUBMIT VALUE=Send>
</FORM>

<% } else
    String elist;
    // really this could come from a database...
    elist=(String)database.getProperty(request.getParameter("group"));
    MailMessage msg=new MailMessage(request.getParameter("From"),elist,
        request.getParameter("Subj"),request.getParameter("msg"));
    SMTP smtp=new SMTP("smtp.myserver.com"); // use yours
    if (smtp.sendMail(msg) == 0) {
%>
    Message sent.
<% } else { %>
    An error occurred. Please try again.

<%
    }
}
%>
</BODY>
</HTML>

```

7.1.8 附件

就 SMTP 而言,所有的邮件消息都是单个实体,然而,可能你已经在邮件消息里看到过文件附件,这是邮件客户端将一个邮件消息解释成多个文件的一种功能。

多部分内容类型 (multipart content type) 是附件的关键。它也允许发送相同消息的另一种表示方式 (如 HTML 和纯文本),下面是一个简单的消息例子:

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="XXXYYYZZZ"
Subject: I have attachments

--XXXYYYZZZ
Content-Type: text/plain

This is the main message.
--XXXYYYZZZ
Content-Type: text/plain
Content-Description: test

This file will appear as test.txt!
--XXXYYYZZZ--
```

该消息的每部分都被两个破折号以及边界文本 (它不能作为消息的一部分出现) 隔开,最后的边界,其前边和后边都有两个破折号。

两个边界之间的文本是独立的消息,可以有自己的题头,尽管题头必须以 Content- 开头 (其他题头在这种情况下没有任何意义)。一个空行将这些子题头和余下的消息分隔开,因为通常文本不一定需要题头,如果你不需要任何特殊题头,可以使用空行来启动附件。在例子里,因为附件的内容描述题头是 test, 内容类型题头是 text/plain, 附件在大多数邮件读者的 test.txt 里都有一个名字。

你可以通过简单的重复—XXXYYYZZZ 分隔符来添加更多的附件 (在所有消息的末尾使用两个破折号)。邮件客户端具体如何表示这些附件要依赖以下的子类型:

- **multipart/mixed**——每部分都是独立的并按照提供的顺序出现。
- **multipart/alternative**——各个部分分别是对相同消息的不同表示 (例如,可能有一部分是文本格式,一部分是 HTML 格式,还有一部分是所有的文档格式,但它们是同样的消息)。
- **multipart/digest**——各个部分都是另一个邮件消息 (经常在邮件列表软件里使用)。
- **multipart/parallel**——各个部分都独立,必须立即出现,当一部分是 HTML 页面,另一部分是在浏览 HTML 页面时要播放的声音文件时,使用这种类型比较有用。事实上,很少邮件程序会这样做,大多数把这种类型看作是 multipart/mixed 类型来对待。

选择分隔符也带有技巧，分隔符必须本身出现在行内，并且不能超过 70 个字符，它前边的回车符和换行符都是边界的一部分，但不是前边的消息的一部分。

注意消息里的每一件事是边界分隔符，每一个边界之前的区域是“无人区”——不能使用足够的信息来解释它，大多数联盟的邮件读者默认情况下会将那里的内容丢弃，有些邮件程序会在这个区域里放入一些指令文本，主要是为那些邮件程序功能不强的用户设置的，并且假设只有他们才能看到那些消息。

附件的常见类型是邮件消息本身（通常是 `message/rfc822` 类型），因为那个消息可能也有附件，所以放入多部分消息是允许的，在这种情况下，每部分必须使用唯一的边界标识。

7.1.9 SMTP 的问题（Twists）

SMTP 的主要问题是任何人都可以连接一个服务器并发送邮件，这个特征反而会帮助那些喜欢散布广告和其他想发送不可跟踪邮件的人，为了避免出现这类事情，很多 SMTP 服务器不再打开，例如，很多邮件服务提供商拒绝来自它们网络外部的连接请求，另一些提供商会从它们的 POP（邮局协议）服务器（见第 8 章）那里维持一个 IP 地址数据库，并对收到的请求进行校验。因为 POP 服务器需要密码，你可以假设来自那个 IP（在一定时间内）的任何请求很可能来自一个授权用户。

这可能使程序试发邮件引发问题，很少服务器需要密码校验，但实际上这些服务器很少见。

7.2 快速解决方案

7.2.1 探寻 SMTP 规范

有关 SMTP 协议的主要 RFC 文档在 RFC2821 里，这篇文档的内容包括几个老的 RFC 文档以及近来的一些实践文档。

7.2.2 连接一个 SMTP 服务器

缺省情况下，SMTP 服务器监听 25 号端口，一旦打开这个端口，就必须等待 220 这个响应码，如果想看看来自服务器的确切的响应码，可以设置正确的服务器名，如下所示：

```
String server="mail.myserver.com";
Socket sock;
sock = new Socket(server, 25 );
BufferedReader reader = new
    BufferedReader(new InputStreamReader(sock.getInputStream()));
System.out.println(reader.readLine());
```

7.2.3 通过 SMTP 发送邮件

使用 SMTP 服务器要求程序向服务器发送一条命令，这个标准的命令是 HELO，但如果想扩展来自服务器的信息和命令，就应该发送命令 EHLO 以取代 HELO 命令，接着可以发送下面三个命令（可以重复这三个命令，以发送多个邮件消息）：

1. MAIL FROM: 从发送方启动邮件
2. RCPT TO: 标识接收者（可能多次发生）
3. DATA: 指定邮件消息

为退出数据模式，在一行的开始发送单个句号。如果该行有句号作为数据，应该发送两个，这样服务器就不会将它作为消息的终止符了，接收者知道如何将行首的两个句号处理作单个句号。

一旦再也没有邮件发送时，可以发送 QUIT 命令，表 7.1 显示了所有可用命令，这里是与一个发送某邮件（同用户输入的行突显）的 SMTP 服务器的手工会话：

```
$ telnet mail.al-williams.com 25
Trying 206.244.69.140 ...
Connected to mail.al-williams.com.
Escape character is '^]'.
220 po2.al-williams.com ESMTP server ready Mon, 14 May 2001 19:06:33 -0400
HELO darkstar@al-williams.com
250 po2.al-williams.com
MAIL FROM: alw@al-williams.com
250 Sender <alw@al-williams.com> Ok
RCPT TO: stamps@al-williams.com
250 Recipient <stamps@al-williams.com> Ok
DATA
354 Ok Send data ending with <CRLF>.<CRLF>
This is a message with 2 lines
And this is the second line
.
250 Message received: 20010514230632.AAA7382@[206.71.104.186]
QUIT
Connection closed by foreign host.
```

消息行不能超过 78 个字符，RFC 坚持认为消息行不能超过 998 个字符（不记入行末的回车符和换行符）。

7.2.4 解释响应码

SMTP 服务器使用类似于 FTP 服务器返回的三位响应码来作响应值，第一位表示消息的类型（2 表示成功，3 表示请求更多信息，4 或者 5 表示失败）。表 7.3 显示了最常见的 SMTP 服务器的响应码。

表 7.3 SMTP 服务器的响应码

响应码	说明
211	系统状态或者系统 help 响应
214	帮助消息
220	<域>服务就绪
221	<域>服务关闭传输通道
250	请求的邮件动作完成
251	非本地用户，需要转交<转交路径>
354	开始邮件输入，以<CRLF>.<CRLF>结束
421	<域>服务不可用，关闭传输通道
450	请求的邮件动作没有进行；邮箱不可得
451	请求的动作被放弃；处理当中有本地错误
452	请求的动作没有进行；系统存储空间不够
500	语法错误，命令不被认可
501	参数中出现语法错误
502	命令没有实现
503	错误的命令次序
504	命令参数没有实现
550	请求的动作没有进行；邮箱不可得
551	非本地用户；请试用<转交路径>
552	请求的邮件动作被放弃；存储空间分配越界
553	请求的动作没有进行；邮箱名不许可
554	事务失败

7.2.5 形成地址

如你所料，构成 SMTP 命令时，可以使用常规的邮件地址。但是，有时除了使用常规邮件地址外，还要显示名字，如果你将一个实际的地址封装在尖括弧里，SMTP 会认可是你为这个地址起的名字，例如：

```
Al Williams <alw@al-williams.com>
```

服务器会将尖括弧外的所有内容都丢弃，除了括弧里的地址，当然，用户看到的内容依赖于题头 From 和题头 To 里边的内容（见下一节）。大多数邮件程序会执行相反的步骤，只显示尖括弧外边的地址。

7.2.6 选择题头

SMTP 作为邮件的传输机制，并不太在意邮件是什么样的格式，但是邮件程序期望邮件有一定的格式，特别地，它们期待一系列题头，即跟在空行和正文文本后边的有关消息的信息。

非标准的题头以 X 开头（例如，X-MimeOLE：由微软 MimeOLE V5.50.4133.2400 设计）。以 Content-开头的题头都是专门为 MIME 指定的消息，在表 7.4 里可以见到常用的题头。

表 7.4 邮件消息的元信息的常见题头

主题	消息主题
To	消息的主要接收者
From	发送者的邮件地址
CC	抄送方式的接收者，这些接收者将出现在邮件地址列表里，并接收到消息的一份拷贝
BCC	暗送方式的接收者，这些接收者会收到消息拷贝，但不出现在地址列表里
Return-Path	接收管理消息的邮件地址
Received	接收的日期和时间
Reply-To	接收任何响应的邮件地址
Date	消息发送的日期和时间
Mime-Version	标识消息为 MIME 类型，并注册消息使用的 MIME 的版本，MIME 题头以 Content-开头
Content-Type	标识消息里的数据类型（如文本/超文本）
Content-Transfer-Encoding	标识消息的编码（如 7 位或者 base 64）
Content-Description	描述 MIME 消息的各部分内容（用于产生附件的文件名）

7.2.7 格式化消息文本

使用 MIME 标准，可以发送 8 位字符的消息、HTML 或者 2 进制数据，如 GIF 文件。具体做法如下：

1. 包含 Mime 版本的题头，将消息标识为 MIME 类型的。
2. 使用 Content-Type 题头来标识文档类型（见表 7.5）。

表 7.5 很常见的 MIME 类型

类型	说明
Application/pdf	Adobe Acrobat
Application/rft	Rich Text Format

续表

类型	说明
application/zip	ZIP 压缩数据
audio/midi	音乐文件
audio/mpeg	使用 MPEG 编码的声音文件
audio/x-realaudio	RealAudio 声音文件
audio/x-wav	Wave 文件
image/gif	GIF 图片
image/jpeg	JPEG 图片
image/png	PNG (Portable Network Graphic) 图片
message/rfc822	邮件消息 (用于消息中的消息)
multipart/alternative	使用不同格式的邮件
multipart/digest	里边包含邮件的邮件
multipart/mixed	包含附件的邮件
multipart/parallel	带有立即显示的附件的邮件
text/html	HTML 文本
video/mpeg	MPEG 编码的视频
video/quicktime	Apple QuickTime 视频
video/x-msvideo	微软 AVI 格式的视频

3. 如果消息的内容里包含有非 7 位的 ASCII 字符, 应该将消息的正文进行编码, 将设置内容传输给编码题头, 以指定用于数据编码的方法。

由于组成附加类型比较容易, 有很多其他 MIME 类型可用, 按约定, 任何常规 MIME 类型应该以 x 开头, 就像 audio/x-realaudio 一样。

7.2.8 使用可引用可打印编码对消息文本编码

如果消息包含非标准字符, 必须使用某种方案将字节转换成 7 位 ASCII 字符对它们进行编码, 其中一种方案就是可引用可打印编码 (内容传输编码: 可引用-可打印)。

这个方法实现起来简单, 但它对那些包含有很多非 ASCII 字符的数据来说效率就不高了, 下面就是使用这个方案对消息进行编码的具体实现步骤:

1. 忽略十六进制值为 21 到 3C 以及 3E 到 7E 之间的字符, 这些都是通常可打印的字符, 除了空格 (20) 以及等号 (3D) 以外。
2. 空格以及制表符 (tab) 可以保持原样, 除非它们位于行末, 那时就必须将它们分别编

码成=20（空格）和=09（制表符）。

3. 所有其他字符（除了回车符和换行符）必须使用三字节的表示方法“=XX”，这里的XX表示字符的十六进制值。例如，要插入一个BEL字符（十六进制为7），就要使用=07。

4. 每行不能超过 76 个字符，可以在回车符和换行符前放入等号将行中断，这样会让接收者将下一行看作是当前行的继续，任何不在等号后边的回车-换行符都可以看作是硬中断。

可以使用等号来对任何字符进行编码，但由于它使用三个字节来表示每个字符，通常不会对那些不需要这种编码的字符进行编码。例如，你可以直接使用大写的 A，=41 也可以表示大写的 A。

7.2.9 使用 Base 64 编码对消息文本编码

大多数用于那些可能包含非 ASCII 字符的消息编码的常见方法是使用 base 64 编码，使用 base 64 编码的步骤如下：

1. 将输入流组织在一起，形成 24 位的 3 字节组（第一个字节的最有意义的位也是新的 24 位字中最有意义的位）。因此 ASCII 文本 ABC 作为 24 位字，其十六进制值为 414243，其二进制值为 010000010100001001000011。

2. 将 24 位字拆成 4 个 6 位组，使用上面同一个例子，将得到 010000 010100 001001 000011（或者在 16 进制里为 10 14 09 03）。

3. 将每个组使用 ASCII 字符来替代（见表 7.2）。

4. 将每行拆分成 76 个字符或者更少（没在表 7.2 里出现的字符都被忽略，因此可以在任何位置插入行中断）。

唯一有困难的地方可能就在文件的末尾，可能要使用一个或者两个空字符来结束，在这种情况下，加 0 来组成整数个 6 位组（因为字节是 8 位的，意味着要有 12 位或者 18 位）。将两个或者 3 个 6 位组转换成 ASCII 字符（像上边一样），使用等号来填充所需要的一个或两个字符，来组成一个四字符的组。

清单 7.5 显示了一个 FilterWriter 的逻辑，这个流使用 base 64 编码将数据写出（写到另一个 Writer 子类），在本章里见到的这个 SMTP 类能接受这个对象类型，用于对正文文本自动编码。

清单 7.5 该过滤器对数据流应用了 base 64 编码

```
// Base 64 encoding
import java.io.*;

public class B64Encoder extends FilterWriter {
    public B64Encoder(Writer w) { super(w); }
    private int charctr=0;
    private int linectr=0;
    private int buffer=0;
```

```
private int encodeByte(int byt) {
    if (byt<=25) return 'A'+byt;
    if (byt<=51) return 'a'+byt-26;
    if (byt<=61) return '0'+byt-52;
    if (byt==62) return '+';
    if (byt==63) return '/';
    return -1; // huh?
}

public void close() throws IOException {
    flush();
    super.close();
}

private void writeit(int c) throws IOException {
    if (linectr++>=76)
        out.write("\r\n");
    linectr=1;
}
out.write(c);
}

public void flush() throws IOException {
    if (charctr==2) {
        buffer<<=2; // make 16 bits to 18 bits
        writeit(encodeByte((buffer&0x3F000)>>12));
    }
    if (charctr==1) buffer<<=4; // make 8 bits to 12 bits
    if (charctr!=0) {
        writeit(encodeByte((buffer&0xFC0)>>6));
        writeit(encodeByte(buffer&0x3F));
        if (charctr==1) writeit('=');
        writeit('=');
    }

    charctr=0;
    out.flush();
}

public void write(int c) throws IOException {
    int[] code = new int[3];
    int i;
    buffer<<=8;
    buffer+=c;
    if (++charctr!=3) return;
    charctr=0;
    for (i=0;i<4;i++) {
```

```

        writeit(encodeByte((buffer&0xFC0000)>>18));
        buffer<<=6;
    }
    buffer=0;
    return;
}

public void write(char[] c, int o, int l) throws IOException {
    while (l--!=0) {
        write(c[o++]);
    }
}

public void write(String s, int of, int l) throws IOException {
    write(s.toCharArray(),of,l);
}

public static void main(String args[]) throws Exception
B64Encoder filter=new B64Encoder(new OutputStreamWriter(System.out));
int x;
filter.write("Hello");
do {
    x=System.in.read();
    if (x!=-1) filter.write(x);
} while (x!=-1);
filter.flush();
}
}

```

7.2.10 格式化多部分消息

MIME 允许将 SMTP 邮件消息分成多个部分，这样做有下述几个原因：

- 多部分可能以不同方式显示相同内容（例如，HTML 和纯文本）。这就是所谓的可替代表示。
- 一个消息可能在不同部分存储着不同的表示元素。例如，一部分可能是文本，另一部分可能是背景音乐或者音频序列，就这就是并行消息。
- 消息的附件应该作为单独的部分出现。这种情况称为混合类型。
- 一个消息可能包含多个邮件消息（就像邮件列表摘要里一样，它会产生名字摘要格式）。

关键是为多部分/可替代（multipart/alternate）、多部分/混合（multipart/mixed）、多部分/并行（multipart/parallel）或者多部分/摘要（multipart/digest）设置 Content-Type 题头。另外，在题头里可以放入逗号，以指定某种特定的边界文本。例如：

```
Content-Type: multipart/mixed ; boundary=":::NEXT PART:::"
```

消息的正文的格式化步骤如下：

1. 以两个破折号和边界文本开头，正文中位于第一个边界之前的任何文本被忽略（有些邮件程序会在这个区域里放入非 MIME 邮件读者指令）。

2. MIME 题头（用于这部分的消息，MIME 题头以 Content- 开头）紧跟在边界文本之后，特别地，还要设置内容类型（Content-Type）。可能也要设置内容描述（Content-Description），很多邮件发送者使用（与内容类型一起）它设置附件的文件名，如果要对正文使用任何特殊的编码方法进行编码，就也可能要设置内容传输编码。

3. 在题头之后，是空行和这一子部分消息的正文。该子部分的末尾产生于边界那一行，包含两个破折号，在这个边界之前的新行是边界的一部分，对消息正文没有影响。

4. 最后一子部分以两个破折号、边界行以及另两个破折号结束。

下面是一个多部分消息的例子：

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="NEXT_PART"
Subject: A Test Message

--NEXT_PART
Content-Type: text/plain

This is the main message.
--NEXT_PART
Content-Type: text/plain
Content-Description: attach1

This is my attachment.
--NEXT_PART--
```

7.2.11 使用 MailMessage 对象

本章的清单 7.1 显示了 MailMessage 对象，可以使用该对象来创建一个邮件消息，也可以在清单 7.3 的类 SMTP 里使用它。这里有关步骤如下：

1. 使用两个构造函数中的一个构造一个对象，必须提供一个 from 和 to 地址，同时还有正文文本和主题，不管使用哪个构造函数。那个长一些的构造函数可以增加 CC（抄送方式的接收者）和 BCC（暗送方式的接收者）列表。

2. 使用 addHeader 方法来增加任何需要的附加题头，如 Mime 版本。缺省情况下，SMTP 将设置标准的题头来标识发送者、接收者和日期。

3. 如果要对正文文本编码（例如使用 base 64 编码），必须设置 encoder 域，使其成为一个类的引用。例如，要使用清单 7.5 里的类 B64Encoder，就要将 encoder 域设成 B64Encoder.class。同时也要设置适合于实际编码的任何题头。

这是一个关于如何创建 MailMessage 对象的代码片段：

```
MailMessage msg = new MailMessage("editor@al-williams.com",
    "alw@al-williams.com", "You are late!", "<H1>Late!</H1>" +
    "<I>Please</I> try to get your tasks done " +
    "<FONT COLOR=RED>on time!");
msg.addHeader("Mime-Version: 1.0");
msg.addHeader("Content-Type: text/html");
msg.encoder = B64Encoder.class;
```

7.2.12 使用 SMTP 对象

如果有 MailMessage 对象，使用清单 7.1 中的 SMTP 对象发送邮件就很容易了，其步骤如下：

1. 使用 SMTP 服务器名或者其 IP 地址，构造一个 SMTP 对象。
2. 调用 sendMail 方法，向 MailMessage 对象传递要发送的内容。
3. 检查返回值，如果值为 0，表明消息发送成功。
4. 如果返回值非 0，可以使用 getLastResponse 得到有关错误的消息。

这是上述工作的一个代码片段：

```
SMTP smtp = new SMTP( "yourserver.domain.com");
rc = smtp.sendMail( msg );
if( rc != 0 )
{
    System.out.println( "Error " + rc );
    System.out.println(smtp.getLastResponse());
}
else
    System.out.println( "OK" );
}
```


第 8 章 POP3 协议

8.1 深入介绍

当我还是个小孩的时候，我的父母经营着一家小店。在那些日子里，电话公司没有竞争，它生产所有的电话，如果没有从它那里租到昂贵的设备，你就不可能将任何东西连接到电话线上，这不过相当于是对马贝尔的潜在的收入损失的一种补偿。

那时我的父母也不想错过生意上的电话，因此他们从 Lafayette（无线电业中的一家小竞争者）那里购买了昂贵的答复机，因为这个机器不能与电话线相连，它成了真正的“乡巴佬”。电话安置在主机盒的顶部，盒子与电话机顶部的撑架相连，里边放着电话听筒，另一个撑架将盒子连向一个普通的磁带式录音机。

那个盒子里有一个比较小的四轨磁带，像你在广播电台里看到的一样，当电话响起来的时候，它会干扰盒子里的某个传感器，盒子会移动支撑架里的一个螺线管，它会将电话移出电话挂钩，四轨磁带会通过手持电话听筒下边的支撑架里的扬声器播放，同时，主机盒会打开录音机（录音机的微型电话在支撑架里边，在手持听筒正下方）。主机盒会将这一状态维持 30 秒钟，接着将所有部件状态重置。

不用说，那时很少有人会拥有答复机，那种机器不仅价格昂贵，而且当有很大噪音时，就会触发它。现在你花 10 美元就可以买到全数字的机器，要比早期的那种机器高级得多。

从某方面来讲，电子邮件是电话的当代版本。简单邮件传输协议（SMTP）规定了如何向另一台机器上的用户发送邮件，当计算机是拥有很多用户的大型机或者小型机时，该协议就已经足够了，问题是现在大多数 Internet 用户都使用 PC 机，机器很可能不是全天 24 小时都在运行，即使都在运行，它也可能不会所有时间都连到 Internet 上，也可能没有一个永久的主机名或 IP 地址。

如果没有处理这个问题的方案，电子邮件在实际应用中就不会有什么用了，这就是为何 POP3（Post Office Protocol 3）协议存在的原因。POP3 允许一个远程计算机（大部分是 Internet 上的计算机）为你保存邮件。

很多 email 客户端，如 Netscape Messenger，微软的 Outlook 以及 Eudora Pro，都使用 POP3 协议，从一个邮件服务器里下载收到的消息。（有一种更新的方案称为 Internet 消息访问协议，也可以采用，但是 POP3 是迄今为止最常用的方案。）你的邮件程序会与一个能够使用 POP3 协议的服务器通信，通过 RFC1939 里陈述的处理方法来取到邮件，而邮件程序仍将使用单独的 SMTP 服务器来发送邮件。

8.1.1 POP3 协议

像很多 Internet 协议一样, POP3 协议要求命令行要以回车符和换行符终止, 缺省的端口号为 110, 该协议使用 TCP 连接, 发送的命令包括三个或四个字符, 服务器不考虑命令的各种情况, 每个命令都可带有参数, 多个参数由空格分开, 每个参数的长度必须在 40 个字符左右。

POP3 协议的响应与其他协议 (如 FTP 或者 SMTP) 不同, 响应的第一个字符将是 “+” 号, 表示一个成功的响应, 或者一个 “-” 号, 表示失败。另外, 服务器将发送一个大写的关键字, 只有一个标准的肯定响应 (+OK), 和一个标准的否定响应 (-ERR), 虽然这个协议能够与其他响应关键字兼容。

POP3 服务器可能有多种状态, 当你第一次连接进来时, 服务器处于授权状态, 客户端必须向它提供信物 (换句话说, 就是提供用户 ID 和密码) 才能做自己要做的事。

一旦提供了正确的信息, 服务器就进入事务处理阶段。它允许客户端取回邮件和有关邮件的信息, 给邮件作标识, 为删除邮件作准备。一般情况下, 服务器在这段时间内会锁住邮箱。最后, 当客户端发送出一个终止会话的命令时, 服务器就进入更新状态, 将清除收到的邮件, 释放邮箱, 并关闭套接字。

POP3 命令不多 (见表 8.1), 其中一些命令在基本实现中不作要求。

表 8.1 POP3 服务器认可的命令

命令	需要 (Y/N)	使用的合法时间段	多行响应 (Y/N)	说明
USER	Y	授权	N	提供用户 ID
PASS	N	授权	N	提供密码
APOP	N	授权	N	发送用户 ID 和对密码进行 MD5 编码的串
QUIT	Y	授权、事务处理	N	终止会话 (如果当前处于事务处理阶段, 则开始更新)
STAT	Y	事务处理	N	引起关于消息数目和消息长度 (字节) 的响应
LIST	Y	事务处理	可能 (Possible)	返回编好序号的消息的状态, 或者如果没有提供序号返回多行响应的状态
RETR	Y	事务处理	Y	取一个带序号的消息
DELE	Y	事务处理	N	为消息作删除标记
NOOP	Y	事务处理	N	无操作

续表

命令	需要 (Y/N)	使用的合法时间段	多行响应 (Y/N)	说明
RSET	Y	事务处理	N	为已有删除标记的消息取消删除标记
TOP	N	事务处理	Y	从一个带序号的消息那里取标题和指定数目的行
UIDL	N	事务处理	可能 (Possible)	为每个消息 (或指定消息) 产生唯一的哈希码, 对于忽略前一个取到的消息比较有用

8.1.1.1 授权

当服务器进入授权状态时, 它发出一个遵守响应格式的提示, 如:

```
+OK POP3 darkstar.al-williams.com v2000.69 server ready
```

从这个例子来看, 客户端可以发送一个 QUIT 命令来放弃, 或者可以使用 APOP 或者 USER 和 PASS 命令进入事务处理状态。

如你所料, USER 和 PASS 命令允许指定用户名和密码, 有些服务器也允许从初始提示符里提取某些文本, 使用密码对其进行加密, 使用 APOP 命令来发送加密过的版本。另外, 服务器也可以提供其他形式的授权 (见 RFC1734)。

注意: 密码是以明文的形式发送的, 除非使用 APOP 命令或者向服务器指定另一种可选的授权方案。

8.1.1.2 事务

事务处理阶段的主要命令就是 STAT、LIST 和 RETR。STAT 命令响应的内容是消息的个数以及邮箱里的字节数。例如:

```
+OK 2 924
```

第一个数就是消息的个数, 第二个数是消息的长度 (以字节计), 如果不想指定数目, 服务器可以发送多行响应:

```
+OK 2 messages (924 bytes)
1 506
2 418
```

当然, 登录到 POP3 服务器的主要原因是获取消息, 这就是 RETR 命令要做的工作, 指定消息数目, 一个多行响应会返回消息文本, 在各行响应里边, 以句号开头的行将以两个句行开始, 以单个句号结束消息。

使用 RETR 命令并没有在服务器上删除消息。为标识一个消息以准备永久删除, 客户端要发送 DELE 命令, 但这并没有真正删除消息, 只是在后来执行 LIST 和 RETR 命令时将消息隐藏了起来, 当服务器更新阶段完成时, 它会真正删除消息, 如果客户端在事务处理阶段中断连接或者发出一个 RSET 命令, 服务器就不会删除带标记的消息。

8.1.1.3 更新

QUIT 命令让服务器进入更新状态, 这是服务器真正删除消息的时候, 同时也解开用户

的邮箱的加锁，很多 POP3 服务器都实现超时处理，以无限地控制邮箱锁。但是，即使超时发生了，服务器也不能删除消息，因为超时状态与更新状态是不同的。

8.1.2 一个 POP3 类

“快速解决方案”一节中的清单 8.2 显示了一个简单的类，能够与一个 POP3 服务器交互，因为最后一章中的 SMTP 类使用了 MailMessage 类来表示邮件消息，使用同一个类来容纳收到的消息。

PopClient 类有一个测试例子 main 方法，从命令行接收输入以确定服务器、用户 ID 和密码，程序简单地打印任何等待邮件的消息（每个消息前都有自己的标题）。

这个类里的代码比较直观，output 方法向输出里增加了回车符和换行符，发给服务器，接着解释服务器的响应。

这个类的真实核心是 readMail 方法，该方法读响应里的所有行，并决定哪些行是标题，哪些行是正文，它正确地将标题汇集在一起，并恰当地填充 MailMessage 对象。

8.1.3 一个常用列表管理器

用来测试邮件处理的最好方法之一就是创建一个简单的邮件列表管理器，列表管理器通常驻留在服务器端，处理收到的消息，用户可以向一个列表地址发送一个邮件消息，管理器将该消息中转到列表的订阅者。另外，用户可以发送消息到订阅邮件的专门的管理员的地址那里，或者从列表里退订。现在使用的最有名的列表管理器之一就是称作 Majordomo 的一个程序。

我曾经创建了一个列表管理器，称作 Seneschal（Seneschal 是中世纪的词，指贵族家庭中的管家或者佣人）。我曾经考虑过使用 Minordomo 或者 Captaindomo 做名字，但它们都用过，我并不是让 Seneschal 成为真正的服务器端程序，而是想在我偶尔手工处理我的邮件时，能借助于它。这就让那些用例代码变得简单，使我集中于离散操作，例如检查邮件、将邮件的地址扫描到列表里、将消息中转到列表订阅者以及删除过时的消息，如果不需要手工运行 Seneschal 程序，可以在调度程序（像 Unix 下的 cron）的帮助下分阶段运行它。

Seneschal 假设你有一个 POP3 邮件帐号并且该帐号可以接收来自多个地址的邮件，例如，我的域名是 al-williams.com，我可以任意的 POP3 有效的应用来连接我所在的域的邮件服务器，即使我使用 alw@al-williams.com 作为我的主邮件地址，我的收件箱还是收集发到该域的任意地址的消息，你可以发送邮件到 bogusmail@al-williams.com，系统照常将消息发送给我，因为我是该域的管理员，为使用 Seneschal，在你的域上指定一个地址作为管理的接口，另一个地址作为列表的接口，这些设置看起来就像是 admin@al-williams.com 和 list@al-williams.com。

程序对只发往这两个地址之一的消息作出响应，并忽略你收件箱中所有的其他邮件，这将让你运行两个（以上） Seneschal 列表成为可能，每个 Seneschal 都带有一个不同的邮件地址，

注意收件箱每次只能被一个程序访问，因此小心不要错误地调度程序，以致它们发生重迭。

程序接受的唯一的命令行参数是属性文件名，它包含了程序的配置选项（见表 8.2），一个选项是另一个文件的名称，该文件拥有订阅者的列表，它们的邮件地址是主键，每个主键的值是一系列可选标志，当前并不使用，但可以用于扩展，以创建功能更强的应用。

表 8.2 Seneschal 程序在属性文件中的选项

参数	意义
addressEmail	主地址列表
adminAddress	管理员地址的邮件列表
smtpserver	SMTP 服务器地址
SMTPPort	SMTP 服务器使用端口（通常是 25）
list	邮件列表属性文件的完全文件路径
popserver	POP3 服务器地址
popuser	POP3 用户名
poppw	POP3 密码

8.1.4 代码

你也可以自己写代码来处理 POP3 和 SMTP 事务，但是有了网上的开放源码库就没有这个必要了，我发现一套 POP3 类，由 John Thomas 编写，非常完整，也容易使用（维护者的站点在 www.geocities.com/SunsetStrip/Studio/4994/pop3.html）。后一节“使用 com.jthomas.pop 包”中的表 8.3 提供了这个 POP3 包的使用方法的梗概，我也使用了第 7 章中的 SMTP 类，Seneschal 程序代码在清单 8.1 中。

这个 POP3 类以一个串数组的形式返回邮件消息。但是，我想使用不同格式的邮件工作可能更简单一些，当 Seneschal 收到一个消息时，它将标题解析成一个 Hashtable 对象。接着，它创建一个包含邮件消息的串（String），程序将调用两个例程之一：一个用作普通邮件，另一个用于管理请求，这要依赖于消息发送到哪一个邮件地址。

使用 POP3 类，取回邮件并不难，基本步骤如下：

1. 构造一个 pop3 对象，用于提供一个 POP3 服务器、用户 ID 以及密码。
2. 调用对象的 connect 方法，建立到服务器的一个连接。
3. 调用 login 方法完成连接。
4. 使用 List 来取回可获邮件列表。
5. 使用 Retr 返回消息的实际文本。
6. 使用 Dele 为将要删除的邮件作删除标记。

7. 调用 Quit 与服务器断连，并处理任何延迟删除的操作。

pop3 对象的方法返回一个 popStatus 对象，不仅可以检查是否成功或失败，也可以得到扩展的结果——例如，消息列表或者消息文本——调用 popStatus 对象的 Responses 方法，返回一个串对象的数组。

当调用 retr 时，会在响应数组里取得邮件消息，每行文本都有一个串 (String)。第一行包括消息的标题 (例如，To: 或者 Subject:)，接着是一个空行，再后边是消息文本。

Seneschal 在两层里使用三个操作方法，第一层的方法 action 带有一个参数 pop3，解析数据，将所有标题存到一个 Hashtable 里。要做到这一点，它就要将串从冒号那里断开，冒号之前 (不包括冒号) 的任何子串就成为哈希表中的键，该行剩余的部分 (也不包括冒号) 成为键对应的值。第一层接着调用第二层中的一个操作方法 (action 或者 admin)，在第一层预告做完所有工作之后，第二层的 action 方法和 admin 方法可以很容易地确定任何特定标题的值。当代码解析碰到空行时，就停下来构造哈希表。

那种简单解析方法对于解析标题工作得很好，但是，admin 方法有一个更大的解析难题，它必须能认出 subscribe 命令和 unsubscribe 命令，甚至当它们嵌到消息文本里的时候也能识别出来。我使用 StringTokenizer 类 (C 语言里看到的 strtok 函数很类似) 来解决这个问题，这个类使用你指定的分界符将串分成若干段。

另一种执行常规解析的例程是 baseEmail，baseEmail 里的代码可以从标题那里找到类似于下面的核心 email 地址：

```
Al Williams <alw@al-williams.com>
```

如果没有 baseEmail，在改变显示名字或者使用不同邮件发送者 (它没有将名字进行相同的格式化) 的情况下，就不能署名。

清单 8.1 Seneschal 类实现了邮件列表管理器

```
// Seneschal e-mail list manager
// Williams
import com.jthomas.pop.*;
import com.al_williams.SMTP.*;
import java.util.*;
import java.io.*;

public class Seneschal {
    String smtpserver;
    Properties maillist; // list of members
    static Properties options; // options
    // helper functions to get addresses
    public static String getAddress() { return getOpt("address", "list@al-williams.com"); }
    public static String getAdmin() { return getOpt("adminAddress", "listadmin@al-williams.com"); }
```

```

public static void main(String arg[])
{
    String cfgfile="maillist.cfg";
    if (arg.length>1) {
        System.out.println("Usage: Seneschal [configfile]");
        System.exit(1);
    }
    if (arg.length==1) cfgfile=arg[0];
    options=new Properties();
    try {
        options.load(new FileInputStream(cfgfile));
    }
    catch (IOException ioe) {
        // can't open setup file?
        System.out.println("Can't open configuration file "+ioe);
        System.exit(9);
    }

    // Everything is ready to go, so start
    new Seneschal().go(arg);
}

// Real main routine
public void go(String [] arg) {
    // set SMTP port
    SMTP.smtpPort=getIntOpt("SMTPPort",25);
    maillist=new Properties();
    // load subscribers list
    try
        maillist.load(new FileInputStream(getOpt("list","maillist")));
    }
    catch (IOException ioe)
        System.out.println(ioe);
        System.exit(2);
    }
    // Set up pop server
    pop3 pop = new pop3(getOpt("popserver"),
        getOpt("popuser"), getOpt("poppw"));
    smtpserver=getOpt("smtpserver");
    popStatus status=pop.connect(), xstatus;
    if (status.OK())
        status = pop.login();
    if (status.OK())
        status = pop.list(); // check mail
        String[] responses = status.Responses();
        String[] xresponses;
        int n;
    // for each message....

```

```

        for(int i=0; i< responses.length; i++) {
            n=responses[i].indexOf(' ');
            n=Integer.parseInt(responses[i].substring(0,n));
// read it
            xstatus=pop.retr(n);
            if (xstatus.OK()) {
                xresponses = xstatus.Responses();
                // check for recipient
                for (int n1=0;n<xresponses.length;n1++) {
                    String to;
                    boolean adminflg=false;
                    if (xresponses[n1].length()==0 ) break;
                    if (!xresponses[n1].substring(0,3).equalsIgnoreCase("to:"))
                        continue;
                    if (xresponses[n1].indexOf(getAdmin())!=-1) adminflg=true;
                    if (!adminflg && xresponses[n1].indexOf(getAddress())!=-1)
                        continue;
// this email message is for me
// do something with email message #n
// including delete it most likely
                    if (action(pop,n,adminflg))
                        pop.dele(n); // kill message
                }
            } else {
                System.out.println("RETR ERROR"); }

        }
        status = pop.quit(); // must quit to delete properly
    }
}

// This is the low-level action that, by default,
// builds a hashtable of the headers and a string of the message
// only override this if you want greater control
public boolean action(pop3 pop, int n,boolean adminflg) {
    boolean header=true;
    popStatus status=pop.retr(n);
    if (!status.OK()) return false;
    String [] responses = status.Responses();
    StringBuffer message = new StringBuffer();
    Hashtable headers = new Hashtable();
    for (int i=0;i<responses.length;i++) {
        if (responses[i].length()==0) {
            header=false;
            continue;
        }
        if (header) {
            int n1=responses[i].indexOf(':');

```

```

        if (n1!=-1) headers.put(responses[i].substring(0,n1).trim(),
                                responses[i].substring(n1+1).trim());
    }
    else {
        message.append(responses[i]);
        message.append("\n");
    }
}
if (adminflg)
    return admin(headers,message.toString());
else
    return action(headers,message.toString());
}

// This is the action routine for normal mail
public boolean action(Hashtable headers, String message) {
    SMTP smtp=new SMTP(smtpserver);
    int rc;
    MailMessage msg;
    message+="\nSent by Seneschal by Al Williams\nTo unsubscribe " +
        "send unsubscribe to " + getAdmin();
    // loop for all members
    msg=new MailMessage(getAddress(),"",
        (String)headers.get("Subject"),message);
    for (Enumeration e=maillist.keys();e.hasMoreElements();) {
        msg.to=(String)e.nextElement();
        rc=smtp.sendMail(msg);
        if (rc!=0) {
            System.out.println("Error " + rc);
            System.out.println(smtp.getLastResponse());
        }
    }
    return true;
}

// This is the admin routine that handles messages to the admin
public boolean admin(Hashtable headers, String message) {
    // commands subscribe, unsubscribe
    boolean rv=true;
    StringTokenizer strtok = new StringTokenizer(message);
    while (strtok.hasMoreTokens()) {
        String tok=strtok.nextToken();
        if (tok.equalsIgnoreCase("subscribe")) {
            rv&=subscribe(baseEmail((String)headers.get("From")));
        }
        if (tok.equalsIgnoreCase("unsubscribe")) {
            rv&=unsubscribe(baseEmail((String)headers.get("From")));
        }
    }
}

```

```
}
return rv; // all commands must succeed or fail
}

// subscribe and send confirming email
private boolean subscribe(String email) {
    boolean rv;
    MailMessage msg=new MailMessage("list",email,"You are subscribed",
        "This message is to confirm that you have " +
        "subscribed to the list\nIf you wish to unsubscribe " +
        "send unsubscribe to " + getAdmin());
    SMTP smtp=new SMTP(smtpserver);
    maillist.put(email,"0");
    if (rv=saveList())
        rv&=smtp.sendMail(msg)==0;
    return rv;
}

// unsubscribe and send confirming email
private boolean unsubscribe(String email) {
    boolean rv;
    MailMessage msg=new MailMessage("list",email,"You have unsubscribed",
        "This message is to confirm that you have " +
        "unsubscribed to the list\nIf you wish to subscribe " +
        "send subscribe to " + getAdmin());
    SMTP smtp=new SMTP(smtpserver);
    maillist.remove(email);
    if (rv=saveList())
        rv&=smtp.sendMail(msg)==0;
    return rv;
}

// Strip email from within <>
private String baseEmail(String email) {
    int n,n1;
    n=email.indexOf('<');
    if (n==-1) return email;
    n1=email.indexOf('>');
    if (n1==-1) return email; //?
    return email.substring(n+1,n1);
}

// Save the list back to property file
private boolean saveList() {
    try {
        maillist.store(new FileOutputStream(getOpt("list","maillist")), "");
        return true;
    }
```



```
    }  
    catch (IOException e) {  
        System.out.println("Can't save mail list: " + e);  
        return false;  
    }  
}  
  
// Get an option with a default value (String)  
private static String getOpt(String key, String def) {  
    String s;  
    s=(String)options.get(key);  
    if (s==null || s.equals("")) s=def;  
    return s;  
}  
  
// Get an option  
private static String getOpt(String key) {  
    return (String)options.get(key);  
}  
  
// Get an integer option  
private static int getIntOpt(String key, int def) {  
    String s=getOpt(key, "");  
    if (s.equals("")) return def;  
    return def;  
}  
}
```

8.1.5 作用

就像它本身一样，Seneschal 类并不很复杂，例如，很多邮件列表程序将发出的消息使用特殊的标题，以阻止邮件回送形成死循环。为外发的消息添加一个唯一的标题是相当容易的，接着你可以检查收到的消息中这种特殊的标题，拒绝提交任何返回的消息。

很明显，对于消息还有其他几种测试，可以拒绝将邮件提交给订阅者列表中没有的地址，也可以拒绝接收有特定词或者附件的邮件。

依据需要，你可以改变 Seneschal 向订阅者发送消息的方法，目前，程序将消息的一个副本发往每个地址，产生多个 SMTP 事务，将每个地址添加 BCC（暗送）列表里，将它们都当作一个 SMTP 事务来处理也许很简单，但是，一个 SMTP 事务的失败可能会阻止所有的接收者来接收邮件。

8.1.6 进一步开发

Seneschal 是首要的列表管理器，但是它的基本结构，特别是方法的基本结构，在更强大

的邮件服务器环境下工作较好。

一旦可以发送和接收邮件，就能够在服务器应用的主机上工作了。可以让用户通过邮件发送请求并作出响应，甚至可以添加应用，自动地答复那些请求价格、股票报价或者新闻故事。当然，你也可以提供传统的邮件服务，如列表服务器、自动响应或者邮件重发等服务。邮件服务要作反馈的另一个地方是在使用邮件服务，的移动设备上，很多传呼机和电话都有收发邮件功能，并且要访问那些设备的用户的唯一方法就是通过邮件发布你的内容。

8.1.7 关于 IMAP

尽管 POP3 协议非常流行，还是有些服务器支持 IMAP 协议（Internet 消息访问协议，由 RFC2060 定义）。IMAP 协议在操作上与 POP3 类似，但有些特殊的特征可以让它适合于远程管理邮箱。

但是 POP3 和 IMAP 依赖于集中式的服务器，它代表用户来接收邮件。POP3 协议要求用户将消息下载到本地机器上，IMAP 也可以这样做，但它也提供操作来管理主机上的邮箱，这就允许在很多不同机器上使用邮件服务，当客户端在本地没有足够空间时，使用 IMAP 也很有用。

虽然 IMAP 协议在某些场合很有用，几乎所有的服务器至少支持 POP3 协议，POP3 更易于理解，因为它使用的命令比 IMAP 少，如果你真的需要 IMAP 支持，你可能会很高兴地发现有一个准备好的类库来处理它（例如，Sun 的 JavaMail）。

8.1.8 使用 JavaMail

因为邮件处理是比较常见的任务，Sun 开发了 JavaMail API。这只是一个规范，定义了邮件客户端对象看起来是什么样子，这些对象的实际操作并没有包含在该规范中，但是 Sun 提供了可重新发布的处理 POP3 和 IMAP（还有 SMTP 用于发送邮件）的实现参考。

因为 JavaMail 包试图处理所有邮件系统，所以到现在为止比你检查过的其他的类都要复杂一些，它不是处理特定服务器上具体的命令，只是从邮件系统里抽象出一些功能对象。

例如，Session 对象与邮件服务器的会话相对应，而 Message 对象是对抽象邮件消息的模型化，但你不会直接使用 Message 类，而是要使用它的一个子类如 MimeMessage，Address 对象允许对消息进行定址，Authenticator 对象表示用户 ID 和密码。

当与一个 POP3 或者 IMAP 服务器相连时，可以通过 Store 对象来访问内容，这个对象将允许你访问 Folder 对象（尽管 POP3 服务器只有一个 Folder，名为收件箱 INBOX）。当然，可以从 Folder 里检索 Message 对象。

使用 JavaMail 没有安装 JavaMail 那么困难，你必须下载 JavaMail 包和 JavaBean Activation Framework（JAF）包。JavaMail 文档有相关解释说明，这意味着你至少要将三个附加的 JAR 文件加到你的 CLASSPATH 里边。

JavaMail 能处理复杂的邮件客户端，但是，一个简单的程序也能实现这个功能。清单 8.3

（在“快速解决方案”一节里）显示了一个客户端程序，与本章的其他客户端程序相当，该程序包含一个 Session 对象，并且通过它还包含一个 Store 对象，通过调用 connect，程序创建了服务器与 Store 对象之间的一个连接，最后，打开一个文件夹查找消息就很简单了。

你将注意到有一行代码，将所有消息的引用附给一个数组，这看起来效率也许不高，但是 JavaMail 的提供者假定直到需要时才从服务器上取真实数据。

8.2 快速解决方案

8.2.1 探寻 POP3 协议规范

POP3 协议描述最多的是在 RFC1939 里，与通常协议一样，在其他几个 RFC 文档里边也有关于 POP3 的扩展和补充。例如，授权方案的扩展在 RFC1734 里就有相关描述。

8.2.2 探寻 IMAP 规范

因为 POP3 协议不适合于操纵管理保留在远程主机上的邮件，有些主机支持不同的协议，即所谓的 IMAP（尽管它们通常也支持 POP3 协议）。IMAP 协议允许在服务器上创建邮件文件夹，并将邮件保留在服务器机器上，邮件客户端接着下载标题和所有它想直接看到的消息，IMAP 有若干个版本，但可以在 RFC1730 上看到最近的版本（第四版）。

8.2.3 解释 POP3 服务器的响应

POP3 服务器在操作成功时总是响应为+OK，如果失败则响应为-ERR。文本总是大写。服务器可能在响应后，会在响应行里置一个空格，后边跟随其他信息（例如，一个操作的结果）。

有些命令产生多行结果，在这种情况下，后来的行就跟随在结果指示行后边。多行结果的末尾将是一个句号。为阻止有些数据过早地终止结果，服务器会向任何以句号开头的行插入一个附加的句号。

如果有一个响应行，对应一个变量名为 line，可以写成下面的代码：

```
// check for end
if (line.length()!=0 && line.charAt(0)=='.' && line.length()==1)
return; // whatever you need to do at the end
// check for escaped period
if (line.length()!=0 && line.charAt(0)=='.' )
line=line.substring(1);
```

8.2.4 使用 POP3 授权

不同的 POP3 服务器可能支持不同类型的授权，但是，基本方法都是使用 USER 命令和

PASS 命令。在连接 **POP3** 服务器之后（通常在端口 110），服务器将发出一个提示响应，接着可以使用 **USER** 和 **PASS** 命令，如下所示（服务器的所有输出都以加号开头）：

```
+OK hello from popgate
USER wd5gnr
+OK password required.
PASS al
+OK maildrop ready, 153 messages (795545 octets) (890161 6291456)
```

8.2.5 了解邮箱状态

可以发出 **STAT** 命令来查询邮箱里包含多少条消息，邮箱占用多少空间，响应将不成功，其中有两个数指示两个数量：

```
STAT
+OK 12 81703
```

这表明有 12 条消息，拥有几乎 82K 的数据，在登录到服务器并缓存结果之后，执行这个命令就很有用：

```
if (!output("STAT")) return false;
StringTokenizer tokens = new StringTokenizer(response);
tokens.nextToken(); // throw away first
messageCount=Integer.parseInt(tokens.nextToken());
totalSize=Integer.parseInt(tokens.nextToken());
output 方法将命令写到服务器，并将结果收集到变量 response 里（完整程序见清单 8.2）。
```

8.2.6 确定消息细节

可以使用 **LIST** 命令来了解一个具体的消息的细节，或者所有消息，当你创建连接时，服务器对邮箱里的邮件任意编号，如果你提供一个消息号，服务器会响应成一行：

```
+OK 1 13185
```

第一个数是消息序号，第二个数是字节数，当然，如果请求的消息序号不存在，就会引起一个提示错误的响应。

```
-ERR invalid message number
```

如果不提供消息序号，服务器会产生多行响应，状态行本身不包含任何消息信息，每个后续的行将对应一个不同的消息，其中包含相对应的两个数。

另一种查找邮件消息的某些信息的方法是使用 **TOP** 命令（尽管不是所有服务器都支持它）。**TOP** 命令允许从消息里获取标题和一定数量的行，例如，要获取第 3 条消息的标题，可以发出如下命令：

```
TOP 3 0
```

这对于只需要显示发送者和主题的邮件程序比较有用，例如，不需要下载完整的消息，当然，服务器会发出多行响应，含有标题、空行、指定行数的邮件消息。

8.2.7 读一个邮件消息

一旦有了要读的邮件消息序号，简单地发出一个带消息号作参数的 RETR 命令就可以了。服务器将响应成消息标题、空行、消息正文，以多行响应的格式出现。

8.2.8 删除一个消息

DELE 命令允许删除指定的消息，但是，消息并没有真正删除，除非成功地发出了 QUIT 命令，如果网络连接中断，服务器实际上不会删除消息，这将有助于防止邮件消息的偶然丢失。

也可以请求服务器在你发送 QUIT 命令之前取消删除，简单地发出一个 RSET 命令，将能有效地恢复你在本次会话中已经删除的邮件。

8.2.9 创建一个 POP3 客户端类

知道了 POP3 协议工作的基本知识，写一个类来处理相关事务就比较简单了(见清单 8.2)，该对象有 6 个重要的域：

- skt——服务器的套接字。
- out——发送数据到服务器的输出流。
- in——来自服务器的输入流。
- response——来自服务器的最后一个响应串。
- messageCount——邮箱里消息的数目。
- totalSize——邮箱大小(字节数)。

output 例程管理到服务器的命令传输以及对响应的解释，返回 false 表明来自服务器的一个错误响应。

connect 方法允许登录到服务器，同时它将 messageCount 和 totalSize 域初始化，其他方法有 readMail 是读邮件消息的一行，readAll 是读一组邮件消息，程序使用来自前一章的类 MailMessage 来表示消息。

清单 8.2 包含一个简单的测试函数 main，因此可以见到类的一些操作，只需在命令行里简单地提供服务器名、用户名、密码，虽然程序显示邮件，但是它并不从服务器上删除它。

清单 8.2 与 POP3 服务器通信的类

```
// POP3 class -- Williams
import java.io.*;
import java.net.*;
import java.util.StringTokenizer;
import java.util.Hashtable;
import java.util.Enumeration;

public class PopClient {
```



```
protected Socket skt;
protected OutputStreamWriter out;
protected BufferedReader in;
public String response;
// mailbox stats
public int messageCount=0;
public int totalSize=0;

// Write a string with CRLF and check response
protected boolean output(String s) throws IOException {
    out.write(s+"\r\n");
    out.flush();
    response=in.readLine();
    if (response.charAt(0)!='+') return false;
    return true;
}

// Connect, login, and set the status variables
public boolean connect(String host,String user,String pw,int port)
throws UnknownHostException, IOException {
    if (port==0) port=110; // default port
    skt=new Socket(host,port);
    out=new OutputStreamWriter(skt.getOutputStream());
    in=new BufferedReader(new InputStreamReader(skt.getInputStream()));
    response=in.readLine();
    if (response.charAt(0)!='+') return false;
    if (!output("USER " + user)) return false;
    if (!output("PASS " + pw)) return false;
    if (!output("STAT")) return false;
    StringTokenizer tokens = new StringTokenizer(response);
    tokens.nextToken(); // throw away first
    messageCount=Integer.parseInt(tokens.nextToken());
    totalSize=Integer.parseInt(tokens.nextToken());
    return true;
}

// Do a quit with or without an RSET
public boolean quit(boolean delete) throws IOException {
    boolean rv=true;
    if (!delete)
        if (!output("RSET")) rv=false;
        if (!output("QUIT")) rv=false;
    try {
        skt.close();
    }
    catch (IOException e) {}
    return rv;
}
```

```

:

// Mark a message for deletion
public boolean delete(int n) throws IOException {
    if (n>messageCount) return false; // catch overflow locally
    return output("DELE " + Integer.toString(n));
}

// Read all messages (could take a while!)
public MailMessage[] readAll() throws IOException {
    MailMessage [] msgs = new MailMessage[messageCount];
    for (int i=0;i<messageCount;i++) {
        msgs[i]=readMail(i+1);
    }
    return msgs;
}

// Read a specific message
public MailMessage readMail(int n) throws IOException {
    if (n>messageCount) return null; // catch overflow locally
    MailMessage mail;
    Hashtable headers=new Hashtable();
    String line;
    StringBuffer bodytext=new StringBuffer();
    boolean body=false;
    if (!output("RETR " + Integer.toString(n))) return null;
    String lastHeader = "";
    // for each line of response...
    while (true) {
        line=in.readLine();
        if (line.length()==0 && !body)
            body=true; // blank line in headers means body
            continue;
        else {
            // check for end
            if (line.length()!=0 && line.charAt(0)=='.' &&
                line.length()==1)break;
            // check for escaped period
            if (line.length()!=0 && line.charAt(0)=='\')
                line=line.substring(1);
        }
        // if body, just build up lines
        if (body) {
            bodytext.append(line);
            bodytext.append("\n");
        }
        else { // headers

```

```

        // if a wrap-around header, append it
        if (line.charAt(0)==' ' || line.charAt(0)=='\t') {
            headers.put(lastHeader,
                (String)headers.get(lastHeader)+" "+
                line.trim());
            continue;
        }
        // otherwise split header line at :
        int ndx=line.indexOf(':');
        if (ndx!=-1) {
            String key=line.substring(0,ndx).trim();
            String value=line.substring(ndx+1).trim();
            lastHeader=key.toLowerCase();
            // and store in hashtable
            headers.put(lastHeader,value);
        }
    }
}

// ok we got it all I think
mail=new MailMessage((String)headers.get("from"),
    (String)headers.get("to"),
    (String)headers.get("cc"),"",
    (String)headers.get("subject"),
    bodytext.toString());

// remove "standard headers"
headers.remove("from");
headers.remove("to");
headers.remove("cc");
headers.remove("subject");
mail.headers=headers;
return mail;
}

// Test main
public static void main(String args[]) throws Exception {
    PopClient pop= new PopClient();
    if (pop.connect(args[0],args[1],args[2],0)) {
        System.out.println(pop.messageCount + " messages");
        if (pop.messageCount!=0) {
            MailMessage [] msgs=pop.readAll();
            for (int i=0;i<msgs.length;i++) {
                MailMessage msg=msgs[i];
                System.out.println(msg.body);
                System.out.println("---- Headers ----");
                Enumeration keys=msg.headers.keys();
                while (keys.hasMoreElements()) {
                    String iKey=(String)keys.nextElement();
                    System.out.println(iKey + ":" + msg.headers.get(iKey));
                }
            }
        }
    }
}

```

```
    }  
    System.out.println("***** NEXT MESSAGE *****");  
  }  
}  
else {  
    System.out.println("Error : " + pop.response);  
}  
pop.quit(false);  
}  
}
```

8.2.10 使用 com.jthomas.pop 包

在 gate.cruzio.com/~jthomas/pop/readme.html 或者 www.geocities.com/SunsetStrip/Studio/4994/pop3.html 可以发现由 John Thomas 写的另一个 POP3 类，它包括了几个（包括源码）可以用来处理收到邮件的类：

- pop3——处理大多数常见邮件操作，它并不支持授权的 APOP 命令。
- apop——扩充了 pop3，并增加对 APOP 命令的支持，如果想使用 apop 类，就同时需要 MD5 类和 MailDigest 类，它们没有源码，来自 Sun 公司。
- testmain 和 testapop——测试两个主类。
- Convert 和 popStatus——执行其他类需要的一些有用函数。

另外，这个包还有几个例子（包括 applet）。

pop3 类使用起来比较简单，可以在表 8.3 中找到使用方法，清单 8.1 中的 Seneschal 邮件列表管理器使用这个类来处理到来的邮件。

表 8.3 对象 com.jthomas.pop.pop3 提供的一些有用方法

方法	说明
connect	连接到 POP3 服务器
login	提供用户 ID 和密码，内部调用 stat 命令
stat	获取邮箱的状态信息
quit	终止与服务器的连接，并处理删除
list	获得有关消息的信息
uidl	找到消息的唯一 ID
retr	获取完整消息
top	获取消息的前部分
dele	为消息作删除标记

续表

方法	说明
rset	取消删除
noop	无操作
get_TotalMsgs	获得等待消息的数目
get_TotalSize	获得邮箱大小
appendFile	获取消息，并将它追加到一个本地文件之后

8.2.11 安装 JavaMail

要在通常的 JDK 安装环境下安装 JavaMail 服务，需要下载两个包：

1. JavaMail 包本身 (java.sun.com/products/javamail/index.html)。
2. JavaBean 激活框架 (Activation Framework) 包 (java.sun.com/beans/glasgow/jaf.html)。

尽管 JavaMail 规范比较抽象，Sun 还是有一个参考实现，可以使用它重新分配 POP3、IMAP、SMTP 服务。

提示：一旦两个包都装了，就需要确认 mailapi.jar 和 mail.jar 两个文件都在 CLASSPATH 里边，另外，如果计划使用上述三个协议，那么将需要 pop3.jar、imap.jar 和 smtp.jar 三个文件，并且 JavaBean Activation Framework 包中的 activation.jar 文件也要使用。

8.2.12 使用 JavaMail Message 对象工作

Message 对象有很多有用的方法，但是，应该承认接收的实际的对象通常是 Message 的子类的实例，因此还有附加的方法，表 8.4 显示了 Message 对象里最常用的一些方法。

表 8.4 Message 对象的一些方法

方法	说明
AddFrom	为该消息添加一组地址到发送者的标题
AddRecipient	为消息添加一个接收者
AddRecipients	添加一组接收者
GetAllRecipients	为消息返回一组接收者
GetFolder	返回包含该消息的文件夹
GetFrom	返回该消息的发送者
GetMessageNumber	返回这个消息的序号
GetReceiveDate	返回消息接收的日期和时间

续表

方法	说明
GetRecipients	返回接收者列表
GetReplyTo	返回对标题的响应
GetSentDate	返回发送日期
GetSubject	返回消息主题
Reply	构造适合于产生响应的一个消息
SetFrom	为该消息设置发送者标题
SetMessageNumber	设置消息的序号
SetRecipient	设置消息的接收者
SetRecipients	设置一组消息接收者
SetReplyTo	设置对地址标题的响应
SetSentDate	设置发送的日期和时间
SetSubject	设置消息的主题

8.2.13 使用 JavaMail Session 对象工作

使用 JavaMail 工作的关键是 Session 对象，它永远不会直接实例化，相反，通常调用 `getDefaultInstance` 返回一个合适的实例来使用。在最小情况下，可以将 `getDefaultInstance` 传递给一个 `Properties` 对象，但是，如果创建了一个新的 Session，它就只使用这个对象，如果想使用缺省设置，那么可以将 `Properties` 对象置空。

但是，也可以设置属性来控制会话的不同方面，例如，`mail.host` 能设置缺省邮件服务器，`mail.user` 能设置用户 ID。因此要通过设置属性对象来设置邮件服务器，可以这样写：

```
prop.put("mail.host", "darkstar.ai-williams.com");
```

8.2.14 在 POP 邮件服务器中使用 JavaMail

在清单 8.3 上可以看到一个简单的程序使用 JavaMail，步骤如下：

1. 创建一个 `Properties` 对象，Session 对象通过 `Properties` 对象来接受选项，在这种情况下，没有选项，但仍然需要 `Properties` 对象。
2. 创建 Session 对象，并不直接对 Session 实例化，而是通过调用 `getDefaultInstance`。
3. 使用 `session.getStore` 来获取服务器的 POP3 存储，也可以指定 IMAP。
4. 使用 `store.connect` 来创建一个从 store 对象到实际服务器上的活连接。
5. 调用 `store.getFolder` 找到 INBOX（收件箱）文件夹（POP3 存储里唯一的文件夹）。对 IMAP 服务器而言，可以访问不同的文件夹。

6. 使用 `folder.open` 以只读方式打开文件夹。
7. 调用 `folder.getMessages` 获取一组消息, 这个数组效率高, 因此它仅当需要时传输数据。

提示: 记住, 可以构造 `Message` 对象, 使用 `JavaMail` 来通过 `SMTP` 发送它, 因此如果现在就使用着 `JavaMail`, 可以考虑用它来发送邮件。

清单 8.3 本例通过使用 `JavaMail` 获取邮件

```
// JavaMail Demo -- Williams
import javax.mail.*;
import javax.mail.internet.*;
import java.util.Properties;

public class JMailDemo {
    public static void main(String args[]) throws Exception {
        String host = args[0];
        String username = args[1];
        String password = args[2];

        // Create empty properties
        Properties props = new Properties();

        // Get session
        Session session = Session.getDefaultInstance(props, null);

        // Get the store
        Store store = session.getStore("pop3");
        store.connect(host, username, password);

        // Get folder
        Folder folder = store.getFolder("INBOX");
        folder.open(Folder.READ_ONLY);

        // Get directory
        Message message[] = folder.getMessages();

        // print first sender, subject, and message
        for (int i=0, n=message.length; i<n; i++) {
            System.out.println(i + ": " + message[i].getFrom()[0]
                               + "\t" + message[i].getSubject());
            System.out.println(message[i].getContent().toString());
        }

        // Close connection
        folder.close(false);
        store.close();
    }
}
```

8.2.15 在 IMAP 邮件服务器中使用 JavaMail

在使用 JavaMail 时, 使用 IMAP 代理 POP 很简单, 从表面来看, 改变 getFolder 的参数就可以使用 IMAP 了。当然, IMAP 的真实技巧是它有很多的功能 (例如, 多个文件夹)。那意味着对 POP 有很多个 JavaMail 方法突然变得有用了。例如, 清单 8.4 显示了一个邮件列表程序, 也显示了 IMAP 服务器上可获得的子文件夹。

清单 8.4 该邮件列表程序使用 IMAP 服务器, 并列出了可用子文件夹

```
// JavaMail Demo -- Williams
import javax.mail.*;
import javax.mail.internet.*;
import java.util.Properties;

public class IMapMailDemo {
    public static void main(String args[]) throws Exception {
        String host = args[0];
        String username = args[1];
        String password = args[2];

        // Create empty properties
        Properties props = new Properties();

        // Get session
        Session session = Session.getDefaultInstance(props, null);

        // Get the store
        Store store = session.getStore("imap");
        store.connect(host, username, password);

        // Get folder
        Folder folder = store.getFolder("INBOX");
        folder.open(Folder.READ_ONLY);
        // print sub folders
        Folder [] folders = folder.list();
        if (folders.length==0) System.out.println("No sub folders");
        for (int ij=0;ij<folders.length;ij++) {
            System.out.println("Subfolder: " + folders[ij].getFullName());
        }

        // Get directory
        Message message[] = folder.getMessages();
        if (message.length==0) System.out.println("No mail");
        // print first sender, subject, and message
        for (int i=0, n=message.length; i<n; i++) {
```

```
        System.out.println(i + ": " + message[i].getFrom()[0]
            + "\t" + message[i].getSubject());
        System.out.println(message[i].getContent().toString());
    }

    // Close connection
    folder.close(false);
    store.close();
}
}
```

第9章 NNTP 协议

9.1 深入介绍

我住在得克萨斯，在选择西班牙的某个住地之前，在得州的大部分地方是不会住得太久的。我们可以收到大量的西班牙电视和无线广播节目，大部分大公司以及政府中介组织都有关于它们的双语信息。

我曾经发现一件有趣的事，那就是任何一门语言，它如果带有常见的拉丁字根，这对讲英语的人来说就太熟悉了。你如果看看法斯（Farsi）语，会发现自己看到一门不同的语言。俄罗斯人甚至使用不同的字母表。另一方面，再快速浏览一下西班牙的报纸，如果想花时间的活，会为报纸中的意思所困惑。

因为罗马人（讲拉丁语的人）征服了他们所知道的世界的大部分，结果他们带去他们的语言并将其传播就不令人惊奇了，甚至当其他语言发生变异时，他们也倾向于以相同的方式进行变异——语言的进化种类。那些比较好的部分就保留下来了，而相对不好的就发生了改变，例如，对所有拉丁语系而言，只有罗马尼亚人保留了名词的多种形式，我所知道的拉丁语言当中也没有保留中性性别。

不像罗马人，Internet 是征服了整个世界。新的协议就像新语言一样，他们倾向于保留那些工作性能好的（精华），抛弃那些工作性能不好的（糟粕）。那意味着很多协议看起来类似，然而实际上它们有着微妙的区别。

对于网络新闻传输协议（NNTP 尤其如此。粗略一看，它几乎与 email 的那些协议完全相同，但是，它们的命令结构不同，而且它们之间还存在着若干细微的差别。

9.1.1 关于 News

News 的目的是允许人们发布和阅读主题多种多样的消息，所有消息都保存在服务器上，服务器将消息组织成组，每一个组与一个主题相对应。

每个组包含的消息与邮件消息格式很类似（见第 8 章）。客户端可以连向包含这些新闻消息的服务器，评阅这些消息，再发布新消息。

除了终端用户，服务器可以互连并且交换消息，其主要思想是用户连接到邻近的服务器，查看新闻消息，这样会减少网上的拥塞和服务器的负载。

RFC977 定义的 NNTP 协议，理论上，对于发送和查看的新闻消息的类型是透明的，但是，系统调整成处理 Usenet 新闻（在 RFC850 上有定义）。Usenet 新闻是一个用于计算机

间交换新闻消息的比较老的系统，它经常使用专用通信链路和电话线来连接，每台计算机的用户能与所有其他用户之间互读消息和互传消息，这比使用 email 向成百上千个用户发送相同的消息要好得多。

Usenet 组带有层次的特征，因此几个火爆的无线节目组可能都在 rec.radio.amateur（例如，rec.radio.amateur.homebrew 和 rec.radio.amateur.satellite）。

9.1.2 NNTP 内幕

从表面上来看，NNTP 与 SMTP 类似，都是发送命令，接收响应码。响应码也是三位数字码，与 SMTP 协议的响应码类似，第一位数确定响应是否成功。

客户端与服务器连接时，通常要提供用户 ID 和密码。客户端可以请求组列表，甚至可以通过提供最后一次读服务器上的组的时标来确定是否有新的组加入。

所有与消息相关的命令都要求客户端发送 GROUP 命令来设置当前组，在组内消息有序列号，它们还有一个唯一不变的 ID。客户端可以向服务器查询消息的 ID，接着下载所有的特定消息或者是下载其中的一部分。

总共只有 15 个基本命令：

- **ARTICLE**——发送该命令可以看当前的文章，也可以用圆括弧带上文章号或消息 ID 来检索文章，文章号只是一个序号，但是消息的 ID 是唯一不变的，如果两个号码都没提供，服务器会认为是当前消息，将其返回给本次会话，但不会递增当前的消息 ID。
- **BODY**——该命令只返回消息的正文，而不返回标题，其他地方与 ARTICLE 命令相同。
- **GROUP**——当你发送 GROUP 命令时，必须包括组名，这就能让服务器知道你要使用特殊的组，如果你不知道合法的组名，可以发送一个 LIST 命令。
- **HEAD**——除了只返回指定消息的标题以外，该命令与 ARTICLE 命令相同，这对客户端只显示标题，让用户选择特定的文章下载，比较有用。
- **HELP**——像很多 Internet 协议一样，你可以以 Telnet 式的终端方式来使用 NNTP，很让人信服。如果要做到这样，HELP 命令会告诉你那些命令是合法的。
- **IHAVE**——允许客户端表明它有一个消息的副本（由唯一的消息 ID 指定）。对那些二级服务器非常重要，因为它们可能拥有主服务器想要拷贝的消息，这个命令不是用于发表新消息的。
- **LAST**——当客户端发送该命令时，服务器将当前文章的指针移向上一篇文章。
- **LIST**——使用该命令可以容纳一组服务器处理的列表，响应的每一行都有一个组名、组的最后一个消息号、第一个消息号以及一个标志。如果服务器接受那一组的消息发表，标志为“y”，否则置为“n”。记住服务器可以接收组的发表，并不意味着它将接收来自客户端的消息发表。
- **NEWSGROUP**——使用该命令，要提供日期和时间，服务器将以新的组列表（比你指定的参数要新）作响应，日期格式可以是 YYMMDD 格式（服务器将会在 50 年

到 2000 年之前加上 1900，这样 88 就是 1988 年，但是 20 指的是 2020)。服务器上用的时间是本地区域时间 (HHMMSS 格式)，除非你在后边指定使用 GMT 格式，这样它会使用格林威治平均时间来求日期和时间。服务器的响应列表以带有一个句号的一行来结束，如果没有新组，就可以只用那个终止行来作响应。

- **NEWSNEWS**——该命令与 **NEWGROUPS** 命令类似，但它为特定组列出新的新闻消息，命令的第一个参数是组（使用*作为通配符），再后边就是日期和时间，其格式与 **NEWGROUPS** 命令相同。
- **NEXT**——使用该命令会使服务器将指向当前文章的指针移到下一篇文章。
- **POST**——发送该命令会使服务器发送一篇文章到当前组里，当然，如果服务器不允许发送，会返回一个错误消息，如果接收到肯定响应，就可以发送一个类似于邮件的消息（就是说，有标题、空行、以句号结束的行）。像邮件一样，如果行以句号开头，必须用两个句号，这样就不会与消息的结束标志混淆。
- **QUIT**——可以想象到，QUIT 命令使服务器断连。
- **SLAVE**——次服务器程序可以发送 SLAVE 命令到主服务器，通知它该程序不面向用户。主服务器并不负责处理这个命令，它只是获取信息。例如，服务器可能会给一个稍高（或稍低）的优先级，相对于用户程序而言，尽管它不负责这样做。
- **STAT**——该命令与 **ARTICLE** 命令语法相同，但是，这个命令不返回任何标题或者文本，相反，它只返回有关指定消息的信息。

每个命令只有几种响应码（见表 9.1），第一位表示分类：1 表示帮助，2 表示成功返回，3 表示需要更多信息，4 表示失败。表中的响应码出现的顺序与服务器可能发送给你的顺序是一致的，例如，如果你想发表一篇文章，服务器可能返回 340，告诉你继续，在你发送完文章以后，服务器会返回响应码 240。

表 9.1 发送到服务器的命令对应的响应码

命令	响应码	意义
ARTICLE	220	文章紧随（跟在文章序号或者 ID 后边）
	412	没有组可选
	420	当前没有文章
	423	在该组没有这个序号的文章
	430	没有这个序号的文章
BODY	222	正文紧随（跟在文章序号或者 ID 之后）
	412	没有组可选
	420	当前没有文章
	423	在该组没有这个序号的文章
	430	没有这个序号的文章

续表

命令	响应码	意义
GROUP	211	选择的组（响应包含估计的文章号，首号，尾号，以及组名）
	412	没有组选择
	420	当前没有文章
	423	该组没有这个序号的文章
	430	没有这个序号的文章
HELP	100	帮助文本紧随（以一个末尾带句号的行结束）
IHAVE	335	发送文章
	435	不发送文章
	235	文章收到
	436	传输失败，再次发送
	437	拒收文章，不再重试
LAST	223	文章收到（跟在序号和文章 ID 后边）
	412	没有组选择
	420	当前没有文章
	422	没有上篇文章
LIST	215	组列表跟随
NEWGROUPS	231	组列表跟随
NEWNEWS	230	文章 ID 列表跟随
NEXT	223	找到文章（跟在序号和文章 ID 后边）
	412	没有组选择
	420	当前没有文章
	421	没有下一篇文章
POST	340	发送文章
	440	不允许发送
	240	发送成功（服务器有可能后来拒绝消息）
	441	发送失败
QUIT	205	退出
SLAVE	202	附属状态提示
STAT	223	文章存在，单独请求文本（跟在文章序号和 ID 后）
	412	没有组选择
	420	当前没有文章
	423	该组没有这个序号的文章
	430	没有这个序号的文章

新闻服务器通常需要用户授权，因为运行一个服务器需要较大的存储量和带宽，如果你真想找到一个开放的服务器，可以在 http://dir.yahoo.com/Computers_and_Internet/Internet/Chats_and_Forum/Usenet/Public_Access_Usenet_Sites/ 上执行一个 Web 查询，打开 NNTP 服务器，在 Internet 上有若干列表，其中包括一个元列表。但是，很多开放服务器由于配置不当，实际上是封闭的。一旦服务器操作员意识到带宽拥塞，他们经常关闭服务器，因此列表经常发生改变。

RFC977 中的基本规范并没有向服务器提供向用户授权的一种方法，但是，几乎所有服务器都实现了 RFC2980 中提到的一种或多种方法。最常见的是，两次使用 AUTHINFO 命令，第一次使用 AUTHINFO 命令，指定用户名；第二次使用该命令，指定密码。例如：

```
AUTHINFO USER alw
AUTHINFO PASSWORD startrek
```

9.1.3 封装 NNTP

像大多数 Internet 协议一样，你可以在余下的程序中写一个类来隐藏协议的复杂性。因为新闻文章看起来像是邮件消息，因而可以使用第 7 章和第 8 章中的 MailMessage 类来放置实际的消息，毕竟，这个类能够处理含有标题消息的基本格式。

在清单 9.1 中可以找到 NewsClient 类，方法使用缺省的 119 端口，等待来自服务器的初始响应，响应码为 200 意指服务器将考虑接受发表，201 表明服务器根本不接受，一旦程序发现响应，它会发送 AUTHINFO 命令，登录到服务器那儿。

清单 9.1 NewsClient 类封装了 NNTP 客户端逻辑

```
// NNTP class -- Williams

import java.io.*;
import java.net.*;
import java.util.StringTokenizer;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Date;

public class NewsClient {
    protected Socket skt;
    protected OutputStreamWriter out;
    protected BufferedReader in;
    public String response;
    public int responseCode;
    public int first=-1;
    public int last=-1;
    public int count=-1;

    protected void write(String s) throws IOException {
```

```
//      System.out.println(s); // debug
out.write(s+"\r\n");
out.flush();
}

// Write a string with CRLF and check response
protected boolean output(String s) throws IOException {
    write(s);
    response=in.readLine();
    //      System.out.println(response); debug
    responseCode=Integer.parseInt(response.substring(0,3));
    return responseCode>=200 && responseCode<=399;
}

// Connect, login, and set the status variables
public boolean connect(String host,String user,String pw,int port)
throws UnknownHostException, IOException {
    if (port==0) port=119; // default port
    skt=new Socket(host,port);
    out=new OutputStreamWriter(skt.getOutputStream());
    in=new BufferedReader(new InputStreamReader(skt.getInputStream()));
    response=in.readLine();
    if (!response.substring(0,3).equals("200")
        && !response.substring(0,3).equals("201"))
        return false;
    if (!output("AUTHINFO USER " + user)) return false;
    if (!output("AUTHINFO PASS " + pw)) return false;
    return true;
}

// Do a quit
public boolean quit() throws IOException {
    boolean rv=true;
    if (!output("QUIT")) rv=false;
    try {
        skt.close();
    }
    catch (IOException e) {}
    return rv;
}

public boolean setGroup(String group) throws IOException {
    if (!output("GROUP " + group)) return false;
    StringTokenizer token = new StringTokenizer(response);
    token.nextToken(); // skip code
    count=Integer.parseInt(token.nextToken()); // estimate count
    first=Integer.parseInt(token.nextToken());
}
```



```

        last=Integer.parseInt(token.nextToken());
        return true;
    }

    // Read a specific message
    public MailMessage readMessage(int n) throws IOException {
        return readMessage(n,"ARTICLE");
    }

    public MailMessage readMessage(int n,String cmd) throws IOException {
        MailMessage mail;
        Hashtable headers=new Hashtable();
        String line;
        StringBuffer bodytext=new StringBuffer();
        boolean body=false;
        if (n==-1) {
            if (!output(cmd)) return null;
        }
        else {
            if (!output(cmd + " " + Integer.toString(n))) return null;
        }
        String lastHeader = "";
        // for each line of response...
        while (true) {
            line=in.readLine();
            if (line.length()==0 && !body)
                body=true; // blank line in headers means body
                continue;
            }
            else {
                // check for end
                if (line.length()!=0 && line.charAt(0)=='.'
                    && line.length()==1) break;
                // check for escaped period
                if (line.length()!=0 && line.charAt(0)=='\')
                    line=line.substring(1);
            }
            // if body, just build up lines
            if (body) {
                bodytext.append(line);
                bodytext.append('\n');
            }
            else { // headers
                // if a wrap-around header, append it
                if (line.charAt(0)==' ' || line.charAt(0)=='\t') {
                    headers.put(lastHeader,
                        (String)headers.get(lastHeader)+" "+
                        line.trim());
                }
            }
        }
    }

```

```

        continue;
    }
    // otherwise split header line at :
    int ndx=line.indexOf(':');
    if (ndx!=-1) {
        String key=line.substring(0,ndx).trim();
        String value=line.substring(ndx+1).trim();
        lastHeader=key.toLowerCase();
        // and store in hashtable
        headers.put(lastHeader,value);
    }
}

// get real message id (header may be blank)
StringTokenizer token = new StringTokenizer(response);
// throw away response
token.nextToken();
headers.put("X-server-number",token.nextToken());
headers.put("X-actual-message-id",token.nextToken());
// ok we got it all I think
mail=new MailMessage((String)headers.get("from"),
                    (String)headers.get("Newsgroups"),
                    (String)headers.get("cc"),"",
                    (String)headers.get("subject"),
                    bodytext.toString());
// remove "standard headers"
headers.remove("from");
headers.remove("Newsgroups");
headers.remove("cc");
headers.remove("subject");
mail.headers=headers;
return mail;
}

public boolean postMessage(MailMessage msg) throws IOException {
    if (!output("POST")) return false;
    String id=response.substring(response.indexOf('<'));
    id=id.substring(0,id.indexOf('>')+1);
    // Note MS Outlook does not like these headers
    // lower case
    write("From: "+msg.sender);
    write("Newsgroups: "+msg.to);
    write("Subject: "+msg.subject);
    write("Message-ID:"+id);
    if (msg.headers!=null) {
        for (Enumeration e=msg.headers.keys();
            e.hasMoreElements();) {

```

```

        String hdr = (String) e.nextElement();
        String val = (String) msg.headers.get(hdr);
        write(hdr+": "+val);
    }
}
write(""); // end of headers
try {
    String line;
    // want to split the input into lines so we can
    // put a CRLF at the end of each
    BufferedReader rdr = new BufferedReader(
        new StringReader(msg.body+"\n"));
    do {
        line=rdr.readLine();
        if (line!=null) {
            // do . escape
            if (line.length()>0 && line.charAt(0)=='.')
                line="."+line;
            write(line);
        }
    } while (line!=null);
}
catch (Exception e) { }
return output(".");
}

public boolean nextPost() throws IOException {
    if (!output("NEXT")) return false;
    return true;
}

public boolean lastPost() throws IOException {
    if (!output("LAST")) return false;
    return true;
}

public NewsClient() { }

public static void readdump(NewsClient news) throws Exception {
    MailMessage msg=news.readMessage(-1);
    System.out.println(msg.body);
    System.out.println("----- Headers -----");
    Enumeration keys=msg.headers.keys();
    while (keys.hasMoreElements()) {
        String iKey=(String)keys.nextElement();
        System.out.println(iKey + ":" + msg.headers.get(iKey));
    }
}

```

```

    }

    // Test main
    public static void main(String args[]) throws Exception {
        NewsClient news= new NewsClient();
        if (news.connect(args[0],args[1],args[2],0)) {
            // Test posting
            MailMessage msg = new MailMessage("test@test.org",
                "alt.test",
                "Java BB test " + new Date().toString(),
                "This is just a test\nPlease ignore");
            System.out.println("Posting = " + news.postMessage(msg));
            System.out.println(news.response);

            // read something
            news.setGroup("alt.test");
            news.output("STAT " + Integer.toString(news.last));
            readdump(news);
            news.nextPost();
            readdump(news);
            news.quit();
        }
        else {
            System.out.println("Error : " + news.response);
        }
    }
}

```

绝大多数命令比较直观,很多程序不得不解析服务器的响应码来选取信息,例如, `setGroup` 方法解析服务的响应码来设置 `count`、`first` 和 `last` 三个域的值。其中 `count` 只是对文章数的一个估计值,另两个域将告诉你服务器拥有当前组的消息的范围。

读消息的方法有一点点曲折,缺省调用 (`readMessage`) 接受一个消息号 (或者是 -1 来读取当前消息)。然而,另有一种方法将命令名作为第二个参数,这允许你发送一个 `HEAD` 或者 `BODY` 命令,这样就只会得到消息的一部分而不是整篇文章。实际上,这对于只获取标题还是有用的,因此可以避免对整个消息的不必要的下载,有了标题可以显示主题行,只下载那些感兴趣的新闻。

新文章可能拥有表示消息 ID 的标题,但某些将没有这种标题,所以 `readMessage` 方法创建一种特殊标题 `X-server-number`,它包含来自服务器的实际消息 ID,而不是消息的标题。另外,它也创建了 `X-actual-message-id` 标题,其中包含消息的序号。

`readdump` 方法只是用于调试错误,示例方法 `main` 使用它来显示某些来自 `alt.test` 组的消

息，当然，这对读取新闻组来说并不是一种很有用的方法。

使用 `MailMessage` 方法发表文章也比较简单，在 `MailMessage` 对象里不是设置接收者的邮件地址，你可以设置新闻组名（由逗号隔开）。方法将从 `MailMessage` 对象那里提取必要信息，并执行发表操作。

9.1.4 Web 上的 NNTP

通过使用 JSP 用户接口与类 `NewsClient` 结合，可以允许用户读取 Web 上的新闻组，在本章的“快速解决方案”一节里可以见到相关例子。

为使问题简单，我写一个前台页面，允许用户选择固定列表中的一个新闻组（见“显示 Web 上的文章”中的清单 9.2）。当然，可以使用 `LIST` 命令来动态产生列表，但是，很多服务器拥有成百上千个组，你很可能不愿意等待。虽然你可以写一个简单的程序来产生 `LIST` 命令输出的本地拷贝，并使用这个本地拷贝来生成选择框，甚至可以安排程序每天自动运行一次或者两次，但是，对这个例子而言，我只使用了一个与 Java 相关的简短新闻组列表。

一旦选择了一个组，`news.jsp` 页面（程序）将产生作用（见清单 9.3）。从 NNTP 的观点来看，这个程序很简单，但是，有些 HTML 产生复杂的结果，问题在于读取消息比较慢。我想通过表格来显示主题、发送者和发表日期（见图 9.1）。但是，受服务器速度影响，读取任何大量的标题可能速度都会比较慢。

Welcome to WNews (comp.lang.java)		
Message	From	Sent
NoClassDefFoundError	alexlang@earthlin...	Tue, 12 Jun 2001 16:10:40 GMT
Re: Runtime.exec() help	"Rish"	Tue, 12 Jun 2001 13:37:15 GMT
Permanent job - E-Commerce Developer Opportunity in Colorado	"David Snow"...	Tue, 12 Jun 2001 15:31:43 GMT
Re: Best IDE for Java	"Claudio Par..."	Tue, 12 Jun 2001 16:04:45 +0200
Re: InetAddress.getByName	Jim Sculley <...	Tue, 12 Jun 2001 09:45:12 -0400
OutOfMemory problem	"JimmyMac" <...	Tue, 12 Jun 2001 13:14:31 GMT
		Tue, 12 Jun 2001

图 9.1 一个 Web 接口（使用 JSP）使程序读取新闻更容易。

能使页面显示速度更快些的一个方法是在读取每篇文章之后调用 `out.flush` 方法，强制数据流到浏览器。如果没有这个调用，可能直到很多消息（甚至是所有消息）准备就绪，浏览器才可能收到数据。但是，使用表格又阻碍实施这个方法，浏览器不会对表格进行格式化，除非它收到了表格里所有行数据。

为了处理这个问题，我实际上将每个消息置于自己的表内，所有表都有固定的宽度，因

此看起来好像是一张表，不幸的是，这产生了另一个问题，表中的列宽依赖于浏览器对回卷文本的适应能力，如果文本没有空格，浏览器将会扩展列，直至合适。当所有消息都成了同一个表的一部分时，这就不会成为问题，如果一个列增长了，所有列都会增长，与它相匹配。但是，使用单个表时，如果某列变宽了，它就不会与其他表的同一列相匹配了。

经过证实，发送者的地址列很麻烦，有些地址可以很长，并且没有空格，因为发送者地址并不是那么关键（他们大多数是随意假冒的），我任意将地址列截短，这样就不太可能溢出表中的对应列。

当你选择一个主题时，news.jsp 文件调用 newsread.jsp 文件（见“读取 Web 上的文章”一节中的清单 9.4）。这个文件实际上取回了完整的消息，并显示它。

两个脚本文件的另一个潜在的问题是在消息部分（看似 HTML 页面）处理相关内容，我写了一个简单的类来对 HTML 文本进行编码（见“读取 Web 上的文章”中的清单 9.5）。正因为大致上所有消息部分都应该进行编码，这样，特殊字符将会得到正确的显示。

由于 news.jsp 和 newsread.jsp 文件需要服务器名、用户 ID 以及密码，所以，我将这些信息放到 newscfg.jsp 文件里边，并且在需要的地方包含这些信息（见“读取 Web 上的文章”中的清单 9.6）。

9.2 快速解决方案

9.2.1 探寻 NNTP 规范

NNTP 规范主要集中在 RFC977 里，像通常一样，NNTP 的内容有扩展，实际上是授权方面，在 RFC2980 里有所定义，几乎是强制性的使用。消息格式，包括所需要的标题，在 RFC1036 里有描述，在 RFC2980 还有一些普通的扩展内容。

9.2.2 连接一个 News 服务器

如果你知道想连接的新闻服务器名或者它的 IP 地址，就可以简单地打开 TCP 套接字开始通信，按常规，119 端口是通常使用的端口，如果服务器考虑到接受 POST 命令，将返回响应码 200，如果它不允许发表，则返回 201。

很少新闻服务器允许不带用户和密码就可以连接，因此常见的响应次序看起来像这样（服务器响应高亮显示）：

```
200 207.218.205.199 DNEWS Version 5.5a3, S1, posting OK
AUTHINFO USER alw
381 PASS required
AUTHINFO PASS startrek
281 Ok
```

当然，如果服务器不需要用户 ID 和密码，就可以跳过 AUTHINFO 命令，继续发送其他

命令到服务器。

9.2.3 选择一个组

在你可以取到任何消息之前，需要选择一个组，使用 **GROUP** 命令不会让人惊讶。要提供组名，相对于该服务器它必须是合法的组（可以使用 **LIST** 命令找到服务器的组）。组名是不区分大小写的，这儿是一个典型的命令及其响应（服务器的响应高亮显示）：

```
GROUP comp.lang.java
211 1049 24145 28225 comp.lang.java selected
```

第一个数是响应码，第二个数是服务器拥有文章数的估计值，虽然这个数是估计值，它要么是正确，要么比实际的文章数大，最后两个数是组中第一个消息和最后一个消息的 ID 号，通常服务器会在这些数前加上 0，该命令产生的一个结果是，它会将当前消息设成服务器中该组的第一个消息。

可以使用 `StringTokenizer` 方法来解析响应行：

```
StringTokenizer token = new StringTokenizer(response);
token.nextToken(); // skip code
count=Integer.parseInt(token.nextToken()); // estimate count
first=Integer.parseInt(token.nextToken());
last=Integer.parseInt(token.nextToken());
```

9.2.4 列出所有的组

可以使用 **LIST** 命令获得服务器支持的所有组的列表，响应将包含组名，每行一个名字，在终止行里还包含一个句号：

```
215 list of newsgroups follows
24hoursupport.helpdesk 426988 348728 y
3b 3370 2825 y
3b.config 26535 25603 v
3b.misc 25953 25151 y
3b.tech 10834 10371 y
.
```

每行包含组名、组中最后一个消息的序号、第一个消息号和一个标志。如果标志值为“y”，服务器将接受该组的发表，尽管对你来说不一定有必要。如果标志值为“n”，服务器将不接受该组的任何发表。

很多服务器认可对 **LIST** 命令的扩展（在 RFC2980 里有相关说明），常见的标志值为“m”，它表示该组是适中的，你的文章发表将被暂停，而不是立即发表。

9.2.5 寻找新组

可以发送 **NEWGROUPS** 命令以知道从某个日期开始是否有新组加到服务器，下面这个会话寻找的是从 2001 年 6 月 2 日 00:00（GMT 时间）开始加入的新组，响应的格式与 **LIST**

命令的响应相同。

```
NEWGROUPS 010602 00:00 GMT
231 list of new newsgroups follows
de.markt.arbeit.biete.misc 48 3 m
de.markt.arbeit.vermittler 140 3 m
de.markt.arbeit.biete.it-berufe 32 3 m
be.comp.networking 22 3 y
..
```

9.2.6 读取文章

一旦选择了一个组，就可以获取属于那个组的任何文章了，可以使用四个命令：STAT, HEAD, BODY 和 ARTICLE。所有命令的响应都很类似，并带有相同的参数。

可以使用三种不同形式的文章获取命令：

1. 可以传递文章序号来获取指定文章。
2. 可以发送文章的唯一 ID（由括弧括起来）来获取文章，注意，如果使用这个方法，就不必选择组了，尽管人们通常都会那么做。
3. 可以不带参数，这种情况下，服务器将返回当前文章。

STAT 命令只响应为一个状态消息。HEAD 命令和 BODY 命令分别返回消息的标题和正文，而 ARTICLE 命令获得消息的所有内容。

对 ARTICLE 命令来说，服务器将发送响应行、标题、空行及正文，HEAD、BODY 和 ARTICLE 响应的最后一行将是只带一个句号的行。

```
ARTICLE 24145
220 24145 <961n$1@news.coriolis.com> article retrieved
From: alw@al-williams.com
Newsgroups: alt.test
Subject: A test
Date: Fri, 9 Feb 2001 16:31:47 +0530
Message-ID: <961n$1 @news.coriolis.com>
This is a test!
.
```

9.2.7 改变当前的文章

当选择组时，服务器会将内部的当前文章指针指到该组的第一篇文章，如果发送了不带参数的 STAT, HEAD, BODY 或者 ARTICLE 命令，这篇文章就是你要使用的文章。

改变当前文章有三种方法：

1. 首先，可以使用 NEXT 和 LAST 命令，因为文章号不一定是相邻的。
2. 也可以使用一个带文章号的命令，会改变当前文章的位置。
3. 如果提供了文章 ID（例如 <961n\$1@news.coriolis.com>），当前文章号不会改变。

9.2.8 查找新文章

如果只对某个日期以后发表的文章感兴趣，可以发送 `NEWNEWS` 命令。该命令需要有新闻组名（可以使用*通配符）以及日期和时间（使用与 `NEWGROUPS` 命令相同的格式）。

注意：该命令可能产生大量数据，实际上，有些站点禁用该命令，因为它产生了太多的数据。

9.2.9 投递文章

可以使用 `POST` 命令向服务器发表文章，产生的响应值为 340（除非有错误产生）。发送的消息是由标题行、空行以及消息正文组成的一个序列。最后一行是一个句号。如果正常的行以句号开始，应该用两个句号来表示，以避免服务器将其作为消息结束符。因为文章会有一个“Newsgroups”标题，在发表之前不必为它选择一个组。

必须提供的标题如下所示：

- **From**——这是你的邮件地址，可以是任何内容，但很多服务器也添加了 `X-Authenticated-User` 标题以标识真实的用户 ID，它可能不是完整的邮件地址。例如，我的地址如果是 `alw@al-williams.com`，那么 `X-Authenticated-User`（当从我的服务器上发表时）可能就是“alw”。
- **Newsgroups**——这个标题指定了要收取消息所在的新闻组，用逗号将组名分隔开一次发送多个组。
- **Subject**——这是新闻读者如何显示你的消息。

下面是一个简单的发表过程的结果（服务器的响应高亮显示）：

```
POST
340 Ok, recommended ID <3b2514bd_2@newsa.coriolis.com>
From:bogus@al-williams.com
Newsgroups: alt.test
Subject: testing
```

```
This is our testÉ
```

```
.
240 article posted ok, wait 3 minutes for it to appear
```

9.2.10 使用 NewsClient 类

清单 9.1 中的 `NewsClient` 类可以简化很多 NNTP 任务，其操作的典型顺序如下：

1. 实例化一个 `NewsClient` 对象。
2. 在 `NewsClient` 对象上调用 `connect` 方法，提供的参数是主机名、用户 ID 和端口号（或者是使用缺省值 0）。
3. 如果希望浏览消息，就调用 `setGroup` 方法。
4. 作 `readMessage`、`nextPost`、`lastPost` 和其他调用，来管理消息。

该类使用第 7 章中的 `MailMessage` 对象来放置消息，也可以通过传送 `MailMessage` 对象给 `postMessage` 方法来发表一个消息。

9.2.11 显示 Web 上的文章

使用清单 9.1 中的类，通过 Web 浏览器来显示和读取文章是可能的，通过使用 JSP 中的 `NewsClient` 对象，任何 Web 浏览器都能访问新闻服务器，清单 9.2 显示了一个简单的页面，允许你从预定义列表选择一个组，它将组名发送给清单 9.3 中的页面 (`news.jsp`)。

清单 9.2 这个 JSP 选择一个组，将它递交给 `news.jsp` 脚本

```
<HTML><HEAD><TITLE>Select a NewsGroup</TITLE></HEAD>
<BODY BGCOLOR=cornsilk>
<H1>Select a news group</H1>
<FORM ACTION=news.jsp METHOD=POST>
<SELECT NAME=group>
<OPTION VALUE=comp.lang.java>comp.lang.java
<OPTION VALUE=comp.lang.java.api>comp.lang.java.api
<OPTION VALUE=comp.lang.java.beans>comp.lang.java.beans
<OPTION VALUE=comp.lang.java.gui>comp.lang.java.gui
<OPTION VALUE=comp.lang.java.programmer>comp.lang.java.programmer
</SELECT>
<INPUT TYPE=SUBMIT VALUE='View Group'>
</BODY>
</HTML>
```

`news.jsp` 脚本完成的工作如下：

1. 实例化 `NewsClient` 对象。
2. 调用该对象的 `connect` 方法。
3. 调用 `setGroup` (设定 `count`、`first` 和 `last` 三个域值)。
4. 读少量消息的标题。
5. 从标题那儿，显示消息的主题，做为到新闻读者脚本的链接。

下面是程序里的一个片段 (有些输出用于示例)：

```
MailMessage msg=news.readMessage(current--, "HEAD");
if (msg==null) break; // whoops?
if (++counter>pagesize) break; // only read some messages
// you could write just the subject here like this: `
out.println(msg.subject);
```

清单 9.3 可以在带有这个脚本的组里看到这些文章

```
<%@ page import="java.util.*" %>
<%@ include file="newscfg.jsp" %>
<%
    int counter=0;
    NewsClient news = new NewsClient();
```



```

if (!news.connect(server,user,pw,0)) {
    out.println("<H1>Can't connect to news server</H1>");
}
else {
    int current;
    int first;
    String firstStr=request.getParameter("first");
    int pagesize=25;
    String nxtfirstStr="-1";
    int nxtfirst;
    String group=request.getParameter("group");
    if (group==null || group.equals("")) group="alt.test";
    out.println("<H1>Welcome to WNews (" + group + ")</H1>");
    news.setGroup(group);
    if (firstStr!=null && !firstStr.equals(""))
        first=Integer.parseInt(firstStr);
    else
        first=news.last;
    current=first;
%>
<TABLE WIDTH=100% BORDER=1 BGCOLOR=cornsilk>
<TR><TD WIDTH=50%><B>Message</B></TD><TD WIDTH=25%><B>From</B></TD>
<TD WIDTH=25%><B>Sent</B></TD></TR></TABLE>
<%
    do {
        MailMessage msg=news.readMessage(current--, "HEAD");
        if (msg==null) break; // whoops?
        if (++counter>pagesize) break; // only read some messages
        String contenttype=(String)msg.headers.get("content-type");
        nxtfirstStr=(String)msg.headers.get("X-server-number");
        if (contenttype==null ||
            contenttype.substring(0,10).equals("text/plain")) {
            String sender=HtmlEncoder.encode(msg.sender.trim());
            if (sender==null || sender.equals("")) sender="&nbsp;";
            if (sender.length()>20) sender=sender.substring(0,17) + "...";
%>
<TABLE BGCOLOR=cornsilk WIDTH=100% BORDER=1>
<TR><TD WIDTH=50%><A HREF=newsread.jsp?group=<%= group %>&art=<%= nxtfirstStr
%>&first=<%=Integer.toString(first)%>><%=      HtmlEncoder.encode(msg.subject)
%></A></TD>
<TD WIDTH=25%><FONT SIZE=2><%= sender %></FONT></TD><TD WIDTH=25%>
<FONT SIZE=2><%= (String)msg.headers.get("date") %></FONT></TD></TR></TABLE>
<%
        out.flush(); // won't really work with a single table
    }
    } while (counter<=pagesize);
%>
</TABLE><HR>

```

```

<%
    nxtfirst=Integer.parseInt(nxtfirstStr);
    if (nxtfirst!=-1 && ++nxtfirst>news.first)
        out.println("<A HREF=news.jsp?group=" + group +
            "&first="+Integer.toString(nxtfirst)+
            ">View earlier messages</A><BR>");
    if (first!=news.last) {
        int nxt=first+pagesize; // guess next number
        if (nxt>news.last) nxt=news.last;
        out.println("<A HREF=news.jsp?group=" + group +
            "&first="+Integer.toString(nxt)+
            ">View newer messages</A><BR>");
    }
}
%>
<A HREF=newsfront.jsp>Pick a new group</A>

```

9.2.12 读 Web 上的文章

清单 9.4 显示了读取三个参数值的脚本 (newsread.jsp):

- Group——组名。
- Art——要读的文章号。
- First——在列表前边显示的第一篇文章的序号。

清单 9.4 当用户选择一篇文章, 这个脚本会将它显示出来

```

<%@ page import="java.util.*" %>
<%@ include file="newscfg.jsp" %>
<%
    String group=request.getParameter("group");
    String first=request.getParameter("first");
    if (group==null || group.equals("")) group="alt.test";
    NewsClient news=new NewsClient();
    if (!news.connect(server,user,pw,0)) {
        out.print("<H1>Can't connect to news server</h1>");
    }
    else {
        news.setGroup(group);
        int n=Integer.parseInt(request.getParameter("art"));
        MailMessage msg=news.readMessage(n);
        out.print("<H1>"+HtmlEncoder.encode(msg.subject)+"</H1>");
        out.print("<h3>From: " + HtmlEncoder.encode(msg.sender)+"</H3>");
        out.print("<h4>Sent: " + (String)msg.headers.get("date") + "</h4>");
        out.print("<PRE>"+HtmlEncoder.encode(msg.body)+"</PRE>");
    }
}
%>

```

```
<A HREF=news.jsp?group=<%= group %>&first=<%= first %>>Back to main News
page</A>
```

参数 `first` 只在用户链接返回到列表页（清单 9.3）并且你需要列表从相同的地方开始的时候才有用，其他两个参数用于标识要读的是哪个消息。

一旦连接到了新闻服务器，代码就比较简单：

```
MailMessage msg=news.readMessage(n);
out.print("<H1>"+HtmlEncoder.encode(msg.subject)+"</H1>");
out.print("<h3>From:" + HtmlEncoder.encode(msg.sender)+"</H3>");
out.print("<h4>Sent:" + (String)msg.headers.get("date") + "</h4>");
out.print("<PRE>"+HtmlEncoder.encode(msg.body)+"</PRE>");
```

Web 浏览器不会将特殊字符解释到消息、主题以及地址行中，因此 `HtmlEncoder.encode` 方法将特殊字符转换成它们在 HTML 中的等价字符（例如，`<`会转换成`<`）。在清单 9.5 中可以找到这个类。

清单 9.5 `HtmlEncoder` 是通用工具类，可以转换特殊字符，使其能够在 Web 浏览器上显示

```
// HTMLEncoder
// only 1 static class that converts things like:
// 3<10 & 4>1 into:
// 3&lt;10 &amp; 4&gt;1

public class HtmlEncoder {
    public static String encode(String intext) {
        if (intext == null) return "";
        StringBuffer out = null;
        char[] orig = null;
        int start = 0, len = intext.length();
        for (int i = 0; i < len; i++){
            char c = intext.charAt(i);
            if (c=='0' || c=='&' || c=='<' || c=='>' || c=='"') {
                if (out == null){
                    orig = intext.toCharArray();
                    out = new StringBuffer(len+10);
                }
                if (i > start)
                    out.append(orig, start, i-start);
                start = i + 1;
                switch (c){
                    default: // case 0:
                        continue;
                    case '&':
                        out.append("&amp;");
                        break;
                    case '<':
                        out.append("&lt;");
                        break;
```

```

        case '>':
            out.append("&gt;");
            break;
        case '"':
            out.append("&quot;");
            break;
    }
    break;
}
}
if (out == null)
    return intext;
out.append(orig, start, len-start);
return out.toString();
}
}

```

news.jsp 脚本 (清单 9.3) 和 newsread.jsp 脚本 (清单 9.4) 都需要服务器名、用户 ID 以及密码, 为将这个信息保存到一个地方, 我将它放到 newscfg.jsp (清单 9.6) 中, 两个文件都使用 include 定向来包含一个到配置数据的引用:

```
<%@ include file="newscfg.jsp" %>
```

清单 9.6 这个脚本为其他需要 NNTP 服务的页面作了配置

```

<%
    String server="news.al-williams.com";
    String user="alwill";
    String pw="bagworm";
%>

```

9.2.13 通过 Web 投递文章

通过 Web 接口发表一个消息, 就是简单地收集足够的信息来填充 MailMessage 对象, 通过 NewsClient.postMessage 方法将它发到服务器上。

在清单 9.7 中可以找到一个例子, 注意那个表单将数据提交给自己, 当脚本发现表单数据时, 它企图将数据传给服务器, 接着随着表单一起显示一个状态消息, 为另一次发表作准备。newscfg.jsp 文件包含配置参数, 就像前面它为清单 9.3 中的 news.jsp 和清单 9.4 中的 newsread.jsp 所做的配置一样。

清单 9.7 使用 JSP 发表文章

```

<%@ page import="java.util.*" %>
<%@ include file="newscfg.jsp" %>

<% String status="";
    if (request.getParameter('group')!=null &&
        !request.getParameter("group").equals("")) {

```

```

        NewsClient news = new NewsClient();
        if (!news.connect(server,user,pw,0)) {
            status="<H1>Can't connect to news server</H1>";
        } else {
            MailMessage msg=new MailMessage(request.getParameter("from"),
            request.getParameter("group"),
            request.getParameter("subject"),
            request.getParameter("msg"));
            news.postMessage(msg);
            status=news.response;
            news.quit();
        }
    }
}
%>

```

```

<HTML>
<HEAD><TITLE>Post to Usenet</TITLE></HEAD>
<BODY BGCOLOR=cornsilk>
<H1>Post to Usenet</H1>
<%= status %><BR>
<FORM ACTION=newspost.jsp METHOD=POST>
<TABLE BORDER=0 WIDTH=50%><TR><TD>
Group:</TD><TD><SELECT NAME=group>
<OPTION VALUE=alt.test>alt.test
<OPTION VALUE=comp.lang.java>comp.lang.java
<OPTION VALUE=comp.lang.java.api>comp.lang.java.api
<OPTION VALUE=comp.lang.java.beans>comp.lang.java.beans
<OPTION VALUE=comp.lang.java.gui>comp.lang.java.gui
<OPTION VALUE=comp.lang.java.programmer>comp.lang.java.programmer
</SELECT>
</TD></TR>
<TR><TD>
From:</TD><TD><INPUT NAME=from>
<FONT SIZE=1>Your email address</FONT>
</TD></TR>
<TR><TD>
Subject:</TD><TD><INPUT NAME=subject><BR>
</TR></TD>
<TR><TD VALIGN=top>
Article:</TD><TD>
<TEXTAREA NAME=msg COLS=80 ROWS=25>
</TEXTAREA>
</TR></TD>
<TR><TD>&nbsp;</TD><TD>
<INPUT TYPE=SUBMIT VALUE=Post>
</TD></TR></TABLE>
</FORM>
</BODY>
</HTML>

```


第 10 章 HTTP 客户端

10.1 深入介绍

静静想一想，你会很吃惊地意识到现在很多人很可能从来就没有用过普通的老式录音机，（另一方面，甚至你就是其中之一）。甚至可以打赌读这本书的人中有一半从来没有接触过 8 磁道的磁带。

对年轻人来说，很难想象音乐不通过 CD 播放，就像我很难想象进入一个无声电影或者在飞船上飞行一样。

在个人电脑流行和 Internet 火爆之前，成为计算机专家有点不可思议，常人都不知道计算机干什么，更不用说要放一台电脑在自己的家里，现在很多家庭都有电脑，并且它们成了访问网络的主要终端。

虽然大多数人会把 Internet 直接想成网络，偏于技术的人知道 Internet 是由于范围广在 Web 里处于领先的，在 Web 出现之前，仍然可以从来自全世界的计算机那里获取信息，但它需要多一点知识和技巧，尽管专家知道 FTP、Telnet 以及其他方法可以访问网络，有很多家庭用户还是认为它们与 8 磁道的磁带放音装置一样古怪。

Web 带到桌面上来的是一种检索和解码来自远程计算机的信息的简单方法，你没有必要知道二进制文件与文本文件的区别，也不必知道如何对文件解码或将多个文件拼接到一起，只需要做点击操作。

从两方面可以完成这项工作，第一，超文本传输协议（HTTP）允许服务器将文件分配给客户端；第二，超文本标识语言（HTML）描述了服务器送往客户端的数据使用 HTML 可以创建到其他文档的链接，这些文档都是通过 HTTP 来传输的。

10.1.1 HTTP 协议

HTTP 协议有几个不同的版本。虽然你应该努力坚持使用最新标准，你还是会发现很多简单的程序使用 RFC1945 中定义的 1.0 标准；有些程序甚至使用 0.9 版中的简单请求，这些不会产生问题，因为服务器尽量保持与以前版本兼容。

HTTP 协议与带有实时连接要求的邮件很相似，想想 Web 浏览器试图读一个网站上的页面的情景，事件发生的简单次序如下：

1. 浏览器打开一个 TCP（传输控制协议）端口（通常是 80），连向服务器。
2. 浏览器发送一个简单请求，例如，“GET/index.htm”。

3. 服务器以真实的 HTML 文档响应。

例如, 清单 10.1 显示了一个访问我的 Web 服务器的 Telnet 会话, Web 服务器采用的是 Apache 作服务器。

清单 10.1 使用 Telnet 手工获取 Web 页面

```
$ telnet www.al-williams.com 80
GET /

<html>

<head>
<title>Welcome to AWC</title>
<meta name="GENERATOR" content="Microsoft FrontPage 3.0">
</head>

<body bgcolor="#C0C0C0" background="new/_themes/capsules/capttext.gif">

<h1>Welcome to AWC</h1>

<p>AWC operates three different Web sites,
please select the one you are interested in:

<ul>
  <li><a href="http://www.al-williams.com/new">AWC</a> - If you are
interested in Windows and Web development, consulting, training, books, and
Al Williams' regular Java@Work columns, this is the page for you. (Our new look
-- under construction)</li>
</ul>

<ul>
  <li><a href="http://www.al-williams.com/awce">AWCE</a> - Looking for
microcontroller development, consulting, our solderless breadboard products,
PAK coprocessors, and our famous Stamp Project of the Month? Here's the
place</li>
</ul>

<ul>
  <li><a href="http://www.al-williams.com/wd5gnr">WD5GNR</a> - Interested in
ham radio or hobby electronics? You'll find a wealth of projects, tips, and
more at Al's Ham Radio site. Check out the Basic Stamp Tip of the Day!</li>
</ul>

<hr>

<p align="right"><small>AWC<br>
310 Ivy Glen Ct.<br>
League City, TX 77573-5953<br>
```


第二个不同在于请求可以带有标题，以提供附加信息，对于 1.0 版，标题完全是可选的，这样，如果你不需要任何标题就可以发送一个空行。但对于 1.1 版（由 RFC2616 定义），必须至少包含一个 host 标题。它标识了服务器计算机，并可能用于提供虚拟主机服务（一台机器用作多个 Web 站点）。同时，除非准备处理一些特殊的 1.1 版协议，应该包括 Connection: close 标题，所以，一个合适的 1.1 版请求会如下所示：

```
GET / HTTP/1.1
Host: www.al-williams.com
Connection: close
```

在最后一个标题之后（这里是 Connection），发送一个空行，服务器就开始响应，如果没有 Connection: close 标题，服务器可能试图让套接字始终打开，并且传输数据也有不同的地方，有些 1.1 版允许的地方，1.0 版却不允许。由于这个原因，除非需要 1.1 版的特征，可能还得用 1.0 版作为一个双边语言，使现在所有的 Web 服务器都一致支持它。

当然，GET 是唯一可用的请求类型，标准中定义了几个请求类型，很多都不是特别流行，下列是可能要遇到的一些请求类型：

- GET——获取文档，数据包括进一个查询串里边。
- HEAD——获取文档的标题。
- POST——随同请求发送数据（从一个表单里），数据作为请求的正文。

另外，HTTP1.1 版允许使用 OPTIONS, TRACE, DELETE 和 PUT 请求，但不会经常碰到这些请求。

10.1.2 状态码

当你作一个请求时（简单请求例外），第一行只返回一个状态，它报告了服务器使用的 HTTP 的版本，它应该不会超过你请求的版本号，也包括一个返回的状态码和消息，例如，在清单 10.2 中将看到下边这一行：

```
HTTP/1.1 200 OK
```

这是一个正常返回，返回的响应码表示服务器发现了文档，并且正将它发送给浏览器，像其他 Internet 响应码一样，可以通过第一位码值来判断三位码值所表示的近似的意思。

起始数为 1 的响应码表示本质信息，200 系列数表示成功的事务，而 300 至 399 会使浏览器重定向到另一页，400 到 599 之间的码值表示有错（例如，当页面找不到时产生的声名狼藉的 404 error）。

表 10.1 中的绝大多数标题（见“快速解决方案”中的“读状态码”一节）看起来都有字面意思。很明显，这时 200 是你想见到的响应码。如果你点击一个中断的链接，就经常见到 404 这个错误码。你可能想知道 401 和 403 两个响应码之间的不同，当响应为 401 时，表示你请求了受保护的资源，需要提供授权信息，通常是指用户 ID 和密码；另一方面，响应为 403 意味着你只是不能获得那个资源，无论怎样尝试。

301 和 302 经常用于将某些存在的链接重定向到新文档甚至是一个不同的网站上，当你

改变域名或者重新设计你的网站时，它就很有用。

100 响应码只在使用 HTTP1.1 时有用，在 1.1 版以前，每个 Web 请求需要一个单独的连接，这并没有发挥好性能，也使得某些事情如维护一个安全连接变得困难了。

如果在请求的标题里不指定 `Connection: close`，服务器会试图将连接保持一段时间，方便你做进一步的请求。为使其工作，服务器不得不以一种特殊的格式发送数据，因为如不这样，就不知道什么时候文档结束。如果一定要使用这种方式，也可以请求服务器，问它是否支持一定的请求，如果支持，它将返回 100 状态码，这样就可以继续发送数据；如果服务器拒绝了你的请求，不必发送数据就可以中断这次事务。

10.1.3 常用标题

服务器为提供有关给客户端的一篇文档的信息，可能会返回任意数量的标题，除了客户端，有些中介计算机，如代理服务器和缓存，也可能要读这些标题并处理它，事实上，有些标题只对代理服务器有用。

由 HTTP1.1 定义的标准的标题包括如下内容：

- `Accept`——表示发送者可接收的数据类型。
- `Accept-Charset`——表示发送者愿意接收的字符集类型。
- `Accept-Encoding`——表示发送者能处理的编码类型。
- `Accept-Language`——表示发送者愿意接收的语言类别。
- `Accept-Ranges`——告诉接收者它可能通过指定一个范围为一部分文档产生请求。
- `Age`——提供了一个缓存文档的估计使用时间（年龄）。
- `Allow`——表示服务器针对该资源能接受的方法（如 GET）。
- `Authorization`——提供用户验证，使文档对于匿名用户不可得。
- `Cache-Control`——通知缓存允许保持文档的备份。
- `Content-Encoding`——允许服务器指定文档使用任意服务器方法进行编码，这可以用于以压缩格式发送数据的情形。（对这种标题感兴趣的用户，见 <http://webwarper.net/ww.pl>）。
- `Content-Language`——指出文档使用的语言。
- `Content-Length`——指出正文部分的大小（以字节计）。
- `Content-Location`——指定文档的另一个可用位置。
- `Content-MD5`——为文档提供一个 MD5 签名（对于检查错误很有用）。
- `Content-Range`——定义了响应里返回的文档的一部分。
- `Content-Type`——表示文档的类型（如 `text/html` 或者 `image/gif`）。
- `Date`——表示服务器发送文档的日期和时间。
- `Etag`——用于区分单个文档的不同变化形式，Etag 的真实值要在具体实现过程中指定，并且允许 Web 服务器使用它自己的规则来辨别这两篇文档是否相同。每个返回

相同精确数据的请求应该带有相同的 Etag。

- Expect——允许发送者通知接收者，它期望一个特别的响应。
- Expires——表示文档的日期和时间由于过期不能用于缓存。
- From——允许浏览器识别产生请求的用户。
- Host——表示一个请求的主机名。
- If-Match——允许发送者将请求与前一个 Etag 匹配。
- If-Modified-Since——提供了与前一个请求的 Date 标题相符的日期和时间，如果服务器确定了文档从该日期以后没有改变，它就返回状态码 304，而不是这篇文档。
- If-None-Match——允许发送者与前一个 Etag 的否定匹配。
- If-Range——允许发送者对文档的特定字节数范围进行判定。
- If-Unmodified-Since——表示 If-Modified-Since 的否定条件。
- Last-Modified——提供文档最后一次修改的日期和时间。
- Location——提供文档（使用重定向）的另一个可用位置。
- Max-Forwards——使用 TRACE 命令来限定提交这个命令的代理服务器的数目。
- Pragma——允许你包含用户标题。
- Proxy-Authenticate——如果需要授权，就伴随代理发出该内容。
- Proxy-Authorization——表示客户端对代理授权请求的响应。
- Range——指定要发送的文档的一部分。
- Referer——尽管拼写有错，它指出了包含到另一个正被请求的文档的链接的文档。
- Retry-After——允许服务器指定何时能够满足一个请求（例如，如果服务器为维护作准备）。
- Server——指定服务器处理请求的类型。
- TE——指定接受请求的传输编码类型。
- Trailer——指定服务器发送响应以后提供的标题。
- Transfer-Encoding——标识文档使用的编码类型。
- Upgrade——指定发送方愿意使用的协议。
- User-Agent——指出有关用户浏览器的信息，与操作系统和其他细节一样。
- Vary——指定什么样的标题可以改变，而不会引起新文档的载入。
- Via——告诉接收者网络路径上的代理或者网关计算机。
- Warning——提供状态码以外的其他信息消息。
- WWW-Authenticate——随同服务器的授权问题一起，提供浏览器。

10.1.4 表单

当你想从 HTML 页面里发送数据到服务器时，不得不使用表单。表单接收用户的数据，并将它作为 HTTP 请求送往服务器，当然，普通的 HTML 页不能处理数据，因此你将希望请

求要适合于 CGI（通用网关接口）程序或者某种脚本。

下面是一个简单的 HTML 表单：

```
<FORM ACTION=http://www.al-williams.com/doForm.jsp METHOD=GET>
Your name: <INPUT TYPE=TEXT NAME=urName><BR>
<INPUT TYPE=HIDDEN NAME=sourceID VALUE=101>
<INPUT TYPE=SUBMIT VALUE=Go>
</FORM>
```

每个表单都有一个 INPUT 域，用户可以往里边填值，编写表单的人也可以使用 SELECT 标签来提供可选择的列表，有些 INPUT 域是隐藏的，例如上例中的 sourceID 域，当你想传递某些信息给服务器，而又不想用户来控制那些数据，使用隐藏域就很常见了。

每个表单都有一个提交按钮（INPUT 域带有一个 SUBMIT 类型），当用户点击 SUBMIT 按钮时，浏览器将数据收集到一起，并将它送到名为 ACTION 属性的带有 FORM 标签所对应的 Web 页面里。

浏览器如何发送数据呢？那要依赖于 FORM 标签中的 METHOD 属性值，在前边的例子里，METHOD 属性是 GET，从而让浏览器将表单数据附加到 URL 上，作为一个查询串，例如，如果用户键入了名字“Boris”，浏览器将产生下面的 URL 请求：

```
http://www.al-williams.com/doForm.jsp?urName=Boris&sourceID=101
```

当然，隐藏域现在在 URL 里是可见的，另外，有些服务器限制你在命令行里传递的数据量，浏览器不得不对包含特殊字符（包括空格）的数据进行编码。

为了避免产生这些问题，大多数表单都使用 POST 方法，当使用 POST 时，浏览器将表单数据作为请求的正文来发送，因此，如果在上例中，将 METHOD 属性值改为 POST，送往服务的请求看起来如下所示：

```
POST /doForm.jsp HTTP/1.0
Content-Length: 24
Content-Type: application/x-www-form-urlencoded
```

```
urName=Boris&sourceID=101
```

在现实中，很少有人需要直接对收到的表单数据进行解码，尽管它对于知道如何形成表单请求常常很有用，在服务器端，通常可以使用 servlet 或者 JSP 程序来处理收到的表单数据，在这种情况下，它们支持的对象会预先为你简化收到的数据。

但是，将数据编码后提交给服务器是很有用的，清单 10.3 显示了一个基于 Java 的接口，它面向美国邮局服务跟踪系统。你可以在命令行里输入一个跟踪号，程序会返回一个 HTML 页面。

清单 10.3 这个 Java 程序将数据提交给邮局服务 Web 站点

```
import java.net.*;
import java.io.*;
import java.util.*;
```

```
public class MailFind {
    public static void main(String args[]) throws Exception {
        System.out.println("Searching for " + args[0]);
        Socket sock=new Socket("new.usps.com",80);
        OutputStream str = sock.getOutputStream();
        InputStream istr=sock.getInputStream();
        String request;
        String data;
        // request data
        data="tracknbr="+args[0]+"\\r\\n";
        // Set up request
        request="POST /cgi-bin/cttgate/ontrack.cgi HTTP/1.0\\r\\n";
        request+="Content-Type: application/x-www-form-urlencoded\\r\\n";
        request+="Content-Length: " + (data.length())+"\\r\\n\\r\\n";
        request+=data;
        // Write it out
        str.write(request.getBytes());
        // For this example, just dump results
        int c;
        do {
            c=istr.read();
            if (c!=-1) System.out.print((char)c);
        } while (c!=-1);
        sock.close();
    }
}
```

向 Web 页提交数据是比较常见的操作，易于封装到一个类里，你也可以在本章的“表单自动提交”一节里找到一个例子（见清单 10.12）。

表单是浏览器能够向服务器传递数据的一种方式，但浏览器也可以通过 HTTP cookies 的机制与服务器进行通信。

10.1.5 Cookies

大饭店的一个特点是里边的职员知道你的姓名，并且能记住你什么时候参观。在预订“普通（一般）”服务而侍者或招待员知道该怎么做的地方，你见过多少？就我个人而言，我只见过几个地方提供那个等级的服务，很可能时代变了，或者碰巧是我遇上了。

最好的网站可以做相同的事情，它们也记得住你，可能它们知道你喜欢什么样的背景颜色，或者知道你热衷于什么体育团体，有些电子商务站点可以记住你的基本信息，因此你没有必要在每个订单上重新键入有关信息。

不幸的是，在 Web 上记住用户的数据可不是一件容易事，每个 Internet 交易是典型的一次性交易，其结果是，如果想识别和存储属于一个用户的相关数据，急需采取各种各样的技术。

10.1.5.1 会话 (Session)

HTTP 协议支持 cookies——服务器可以存放从用户浏览器那里获取的少量数据。如果服务器附给每个浏览器一个唯一的标识,它就能够识别来自同一个浏览器的重复请求。(也有其他方法来创建这种联系,但是 cookies 是最干脆而且最常见的方法。)浏览器不必接受 cookies,但因为如此多的站点需要 cookies,大多数用户并没有完全将 cookies 拒之门外。

当服务器需要浏览器记住什么东西时,它在标题里发送一个 cookie,例如:

```
Set-Cookie: ASPSESSIONIDQQQGGJMO.CHNHAFMDHAOKPENB; path=/
```

当客户端产生一个对设置了 cookie 的服务器的请求时,它将在 cookie 的标题里包含数据:

```
Cookie: ASPSESSIONIDQQQGGJMO= CHNHAFMDHAOKPENB
```

Cookie 在哪儿存放要依赖于浏览器, Cookie 可以有使用的截止日期,在此情况下,浏览器很可能将它们存放到一个文件(或者多个文件)里,没有截止日期的 cookie 在浏览器打开时一直有效,这样它们很可能在别的地方也具有记忆功能。

像很多类似的系统一样, JSP 人为地创建了 session 的概念,浏览器第一次请求一个页面时,服务器会附给浏览器一个唯一的 ID,并将它存到客户端的 cookie 里,后边的请求将包含这个 cookie,这样服务器就知道哪些请求来自同一个浏览器。

Session 管理在很多情况下比较重要,例如,当顾客填满一个购物车时,零售商需要记住顾客访问该站点的整个期间内购物车里的内容,但是, session 管理功能只达到这个层次了,浏览器只要愿意,可以随时删除 cookies,最后, cookies 也会过期,典型的,一个 session 的 cookie 会持续 30 分钟(每次页面的刷新会重新启动这个时钟)。很难见到一个 session 会持续几天或者几个小时,并且, cookie 是属于特定计算机的特定浏览器的,你上班时用的 cookie 不会适用于你家里的 session。

存储 session 数据的一个解决方法是将它存到一个持久的表单里,像数据库存放数据一样。还有一种方法是让用户来识别他们自己,可以在每个用户的基础上记起他们的偏爱,而不只是为了 session。

10.1.5.2 Beans 和 JSP

JSP 有很多你需要的功能来处理这种情况,但是,它的逻辑有点复杂,可能全部使用 JSP 来写代码很容易混淆,相反,我决定创建一个用户属性数据库——JavaBean,可以在 JSP 里很容易地调用它,核心的 bean 比较抽象——具体实现可以将数据存储到一个普通文件、属性文件、XML(扩展标识语言)文件、数据库文件或者任何你想存储数据的地方。Bean 能让你很容易地将有密码保护的用户名和一个属性集相联系。

JSP 是创建 servlet 的一种简化方法,大多数人都同意创建一个 JSP 脚本比从头写一个 servlet 要简单些,但是,在 JSP 里写太多的 Java 代码是一个非常坏的习惯, JSP 的一个强项在于它让 HTML 的设计者将精力集中在网页结构上,并在需要的地方写上 Java 代码,但是,当代码淹没了 HTML 标识的时候,这个优点就荡然无存了。

幸运的是,有一个折中的办法,可以写一个 JavaBean,让你很容易地将它嵌到 JSP 页面

边，不要担心 JavaBean 如何复杂，你会发现创建 JavaBean 很容易，它们与普通的符合一些命名规范的 Java 对象差不多。事实上，你根本不用创建 JavaBean，但是，JSP 对于包含 beans 特别熟练，创建一个 beans 造成的困难是极小的。

什么才算是 JavaBean 呢？为便于讨论，一个 bean 是任意的公有 Java 类，它不会暴露自己的公有域，那并不意味着对象不能有域，或者对象不能与外边的程序以某种方法共享域值。不是使用域，beans 拥有属性。属性看起来与域很相似，但是，当你要读或者改变一个属性时，你实际上是在 bean 的范围内调用特殊的函数来工作，如果想要一个名为 xyz 的属性值，就得提供 getXyz 方法和 setXyz 方法，很多有 bean 意识的程序当你请求访问一个属性时，自动地产生这些方法调用，有 bean 意识的程序知道它确实是一个属性。

当然，get 和 set 方法可能只是提供对非公有域的访问，另一方面，有些属性并不直接和一个变量相对应，例如，可能有一个域叫 isServerUp，它能指出另一台计算机在网上是否可以访问，而 getIsServerUp 方法可能不会反映一个变量。相反，它将要看看运行的服务器是否使用套接字，setIsServerUp 方法可能不存在，因为在这种情况下它没有什么意义，属性是只读的。

那就是 JavaBeans 所有的内容吗？不，但是对于向 JSP 添加功能而言，那就是你真正需要了解的。

10.1.5.3 设计

属性数据库的基类是 UserDB（见清单 10.4），该类提供了抽象方法，用于处理实际的数据传输，步骤如下：

- GetProperty——为当前用户从数据库中获取属性值。
- WriteProperty——为当前用户写一个属性值（不必保存）。
- Save——保存属性数据库到一个永久存储区。
- Load——为一个指定的用户（校验密码）装载数据库。
- Create——创建一个新用户。

清单 10.4 这个基类允许记忆 session 之间的用户属性

```
// UserDB object
import java.io.IOException;
import javax.servlet.http.*;

abstract public class UserDB implements HttpSessionBindingListener {
    protected String user;
    protected String dir;
    protected boolean autosave = false;
    protected boolean dirty=false;
    public boolean getAutosave() { return autosave; }
    public void setAutoSave(boolean b) { autosave=b; }
    public String getUser() { return user; }
```



```

public void setUser(String user) { this.user=user; }
public String getDir() { return dir; }
public void setDir(String dir) { this.dir=dir; }
abstract public String getProperty(String key,String deflt);
abstract public void writeProperty(String key,String value);
abstract public void save() throws IOException;
abstract public boolean create(String password) throws IOException;
abstract public boolean load(String password);
// methods to manage sessions
public void valueBound(HttpSessionBindingEvent event) {
    // no action
}
public void valueUnbound(HttpSessionBindingEvent event) {
    if (autosave && dirty) {
        try {
            save();
            dirty=false;
        }
        catch (IOException e) { }
    }
}
}

```

尽管抽象的基类不提供任何功能，但它提供了对这些功能的支持，特别地，基类维护了用户名、识别数据位置（不一定是实际中的目录名——它可能是一个数据集名）的目录前缀、一个 autosave 标记、一个脏标记。如果你设置了 autosave 标记，并且数据库是脏的（意思是说，数据库发生了改变），基类程序在会话（session）过期时将自动保存数据。

为检测 session 是否过期，基类必须实现基本的二方法接口 HttpSessionBindingListener，这个接口能让你发现 session 何时开始，何时结束。

通常，session 开始于用户第一次浏览网页的时候，过了一定的时间以后，如果用户没有再去激活网页，或者你的脚本明确地让 session 无效，session 就会过期。你可以使用下述代码来释放激活的 session：

```
request.getSession().invalidate();
```

因为抽象的基类完全实现了 HttpSessionBindingListener，没有必要让继承类知道有关 session 管理的任何事情，所有的工作发生在幕后。

10.1.5.4 实现

关于用户信息的一个可能的数据库是普通 Java 属性文件的集合，我喜欢使用属性文件充当 Java 程序中简化的、易于理解的数据库，当然，对于一些大型工业企业网站而言，需要使用更好的数据库，但是属性文件对于一个适中的网站而言使用起来比较容易，示例程序也做得很简单，基类没有任何偏好，你可以很容易地继承一个新类，使用你自己的数据库。

清单 10.5 显示了属性文件的实现，你将看到只有一些实质性方法，分别是 create, load, save, getProperty 和 writeProperty。如你所料，这些不同的方法都映射到受保护的 Properties 对象

的一个实例，每个用户都有他（她）自己的属性文件，它们都存放到指定的目录里，密码在属性文件里是未加密的，因此目录不应该放到 Web 服务器能够访问的地方。

清单 10.5 该类使用一个属性文件存储用户信息

```
// UserDB that uses Property files

import java.util.*;
import java.io.*;

public class UserDBProp extends UserDB {
    protected Properties prop;
    public boolean create(String password) throws IOException {
        FileInputStream istream;
        try {
            istream = new FileInputStream(
                dir + "/" + user + ".properties");
            istream.close();
        }
        catch (FileNotFoundException e) {
            // ok to create
            prop=new Properties();
            prop.put("password",password);
            save();
            return true;
        }
        catch (IOException e) {
            return false; // some error?
        }
        return false; // can't create -- already exists
    }
    public boolean load(String password) {
        FileInputStream istream;
        prop=new Properties();
        try {
            istream = new FileInputStream(
                dir + "/" + user + ".properties");
            prop.load(istream);
            istream.close();
        }
        catch (IOException e) {
            prop=null;
            return false;
        }
        // check password
        if (!password.equals(prop.getProperty("password"))) {
            prop=null;
            return false;
        }
    }
}
```

```
    }  
    return true;  
}  
  
public void save() throws IOException {  
    FileOutputStream os=new FileOutputStream(  
        dir + "/" + user + ".properties");  
    // use save if JDK<1.2  
    prop.store(os,"WebProperties");  
    os.close();  
}  
  
public String getProperty(String key, String deflt) {  
    if (prop==null) return deflt;  
    String rv=prop.getProperty(key);  
    if (rv==null || rv.equals("")) rv=deflt;  
    return rv;  
}  
public void writeProperty(String key, String val) {  
    if (prop==null) return;  
    dirty=true;  
    prop.put(key,val);  
}  
}
```

缺省情况下, `autosave` 的特征是关闭的, 因此你必须在想要更新任何底层属性文件时调用 `save` 方法, 也可以打开自动保存功能, 这样当 `session` 结束时, 文件也更新了。

创建 `bean` 最难的一部分是如何让 `JSP` 容器能够认识它, 这就需要将你的 `beans` 放到服务器的类的路径 (class path) 里, 到底将类文件放到何处要依赖于你使用什么容器以及你的特定安装。一般情况下, `bean` 的类文件必须位于 `JSP` 应用目录下的 `WEB-INF/classes` 的子目录中。

使用 `beans` 需要一些页面, 不管是什么样的页面。首先, 需要一个登录页来接受用户 `ID` 及其密码, 接着, 很可能需要一个页面, 让你创建一个新 `ID`。用一个页面来终止用户的 `session` 也很方便, 可有效地记录离开系统的用户。

使该系统可行的一个关键是使用 `session` 生命期来创建数据库 `bean`, 当 `JSP` 页里包含一个 `bean` 时, 可以指定生命期以及你想要使用的名字, 每次包含 `bean` 时, `JSP` 容器首先要检查是否已经有 `bean` 在那个生命期, 如果有, 它只是取得那个 `bean` 的实例, 如果没有, 容器会为你创建一个新的 `bean`。

带有 `session` 生命期的 `beans` 会持续到会话结束, 那意味着如果你使用 `session` 生命期创建一个数据库 `bean`, 每页的用户访问将使用相同的 `bean`。你只需在登录页对 `bean` 进行初始化, 而不用在每个后继的页对它进行初始化。因为每个用户都有独立的 `session`, 你不会与访问一个不同用户设置的用户之间产生隐私冲突。当然, 这个简单的系统根本谈不上安全, 你需要避免存储敏感信息, 例如信用卡号等, 放到一个明文的属性文件中的情况应该回避。

创建或者查找 bean 的 JSP 标签如下所示:

```
<jsp:useBean id="upd" class="UserDBProp" scope="session"/>
```

这个 bean 的名字在这里叫做 upd, 它与下列表达方式类似:

```
UserDBProp upd = new UserDBProp();
```

但是, 很大的不同在于如果 upd 在会话期间已经存在, 你只须找回现有的该类的实例。

你也可以使用特殊标签来访问 bean 的属性 (尽管你可以在普通的 JSP 脚本里引用它们), 例如, 下面是设置 dir 属性的一个标签:

```
<jsp:setProperty name="upd" property="dir" value="c:/tmp"/>
```

不要忘了收到一个 setDir 调用时实际 bean 的结果, 这个标签并不试图直接访问 dir 域, 即使看起来好像是这样。

在清单 10.6 里可以找到一个登录的脚本示例。这个表单将数据提交给自己, 如果它探测了数据入口, 就试图将用户的属性载入。如果探测失败 (也就是说, 载入数据的方法返回一个 false 值), 那么相同的表单会随同一个附加的红色字体的错误消息一起显示。如果载入成功, 脚本会自动将页面重定向到主页那里 (见清单 10.7)。

清单 10.6 这个脚本要求用户登录

```
<jsp:useBean id="upd" class="UserDBProp" scope="session"/>
<jsp:setProperty name="upd" property="dir" value="c:/tmp"/>
<%
    boolean err=false;
    if (request.getParameter("uid")!=null) {
        upd.setUser(request.getParameter("uid"));
        if (upd.load(request.getParameter("pw"))) {
            response.sendRedirect("uphome.jsp");
        } else {
            err=true;
        }
    }
%>

<HTML>
<HEAD><TITLE>Please log in</TITLE>
</HEAD>
<BODY BGCOLOR="cornsilk">

<% if (err)
    out.print("<FONT COLOR=RED>Invalid ID or password. Please try again</FONT>
<BR>");
%>
<TABLE>
<FORM METHOD=POST ACTION=updlogin.jsp>
<C<TR><TD>
UserID: </TD><TD><INPUT NAME=uid></TD></TR>
```

```

<TR><TD>
Password:</TD><TD> <INPUT NAME=pw TYPE=PASSWORD></TD></TR>
<TR><TD>&nbsp;</TD><TD>
<INPUT TYPE=SUBMIT VALUE="Login"></TD></TR>
</FORM>
</TABLE>
<BR><A HREF=updcreatc.jsp>Need a user ID?</A>
</BODY></HTML>

```

清单 10.7 这个简单的主页通过用户名来欢迎用户

```

<jsp:useBean id="upd" class="UserDBProp" scope='session' />
<jsp:setProperty name="upd" property="dir" value="c:/tmp" />
<HTML>
<HEAD><TITLE>Home Page</TITLE>
</HEAD>
<BODY BGCOLOR=cornsilk>
<P>Welcome!
<% if (upd.getProperty("name","").equals("")) {
%>
unregistered user.
<% } else { %>
<%= upd.getProperty("name","") %>
<BR>
<A HREF=updlogout.jsp>Log out</A>
<% } %>
</BODY>
</HTML>

```

创建一个新用户实际上操作相同（见清单 10.8），唯一的不同是这种情况下，新用户不能是已经创建的用户，同时，在成功地创建用户后，脚本创建一个属性文件，并将用户重定向到登录页。

清单 10.8 该 JSP 创建一个新的用户 ID

```

<jsp:useBean id="upd" class="UserDBProp" scope="session" />
<jsp:setProperty name="upd" property="dir" value="c:/tmp" />
<%
String errorMessage="";
boolean err=false;
if (request.getParameter("uid")!=null) {
if (!request.getParameter("pw").equals(request.getParameter("pw2"))) {
errorMessage="Passwords did not match";
err=true;
}
if (request.getParameter("pw")==null ||
request.getParameter("pw").equals("")) {
errorMessage="Password must not be blank";
err=true;
}
}

```



```

    }
    if (!err) {
        upd.setUser(request.getParameter("uid"));
        if (upd.create(request.getParameter("pw"))) {
            upd.setProperty("name", request.getParameter("uname"));
            upd.save();
            response.sendRedirect("updlogin.jsp");
        }
        errormessage="User ID in use.";
        err=true;
    }
}

%>
<HTML>
<HEAD><TITLE>Creating new ID</TITLE>
</HEAD>
<BODY BGCOLOR="cornsilk">

<TABLE>
<% if (err) { %>
<BR>
    <FONT COLOR=RED SIZE=3><%= errormessage %></FONT>
<BR>
<% } %>
<FORM METHOD=POST ACTION=updcreate.jsp>
<TR><TD>
Name: </TD><TD><INPUT Name=uname></TD></TR>
<TR><TD>
UserID: </TD><TD><INPUT NAME=uid></TD></TR>
<TR><TD>
Password:</TD><TD> <INPUT NAME=pw TYPE=PASSWORD></TD></TR>
<TR><TD>
Verify password:</TD><TD><INPUT NAME=pw2 TYPE=PASSWORD></TD></TR>
<TR><TD>&nbsp;</TD><TD>
<INPUT TYPE=SUBMIT VALUE="Create"></TD></TR>
</FORM>
</TABLE>

</BODY>
</HTML>

```

10.1.5.5 改进

当然，对这个系统可以作出的一个明显的改进是使用真实数据库，如 MySQL 或者 Access，来存储数据。改变的数据安全也不失为一种好方案。另一个有用的特征是为以后用户登录时提供一个 cookie，记住用户的 ID 和密码，或者为用户提供一种方法，在用户丢失密

码时通过邮件找回密码。

你甚至可以使用这个系统来实现一个基本的安全系统，每页可以包含一个 JSP 文件来检查用户的登录是否已经被接受。如果不是，它可以将用户页面重定向到登录页面，这将使得不输入密码就浏览一个页面成为不可能的事情。这个主题的变化之处在于附给用户一个安全关口，以控制他们能看的内容。某些页面可能访问不到，除非你的安全关口是畅通的。

10.1.6 Applets (小程序)

Java 在 Web 上如此流行的一个原因是因为安全性，因为 applets 运行在浏览器的背景下，浏览器可以严格控制 applet 哪些能做，哪些不能做。虽然这对用户来说是一个福音，但对于开发人员来说，它有些限制得过死。

当然，你可以对你的 applets 进行数字签名，以希望用户自己来选择是否放松安全限制，但是这不是一种理想的解决方案，最好的解决方法是寻找一种在现存的安全框架里就能工作的方法。

例如，假设你想写一个需要存储数据的 applet，那么你的选择很少，你不能在客户端机器上写一个文件。但是，你可以打开一个面向服务器的网络连接，将文件写到服务器上，这需要服务器上的一个程序来监听 applet，并处理它的请求。

接着有另一个问题：你可以打开只面向你自己的服务器的网络连接，例如，这可以阻止自己的机器恶意的 applet 发送邮件。但是如果你想让两台不同机器上的 applets 通信，就会出现问题，例如，一个多角色游戏，不能简单地连接两个 applets。服务器必须充当电话的交换机，将一个 applet 的数据转到另一个 applet。

我会向你展示如何创建一个简单的服务器，用来让两个 applets 互相通信。示例 applets 让两个角色在网上玩 tic-tac-toe 的游戏（或者是像我的英国朋友玩○和叉的游戏那样，○和叉是棋中的两类符号）。在图 10.1 中可以见到游戏的处理过程。

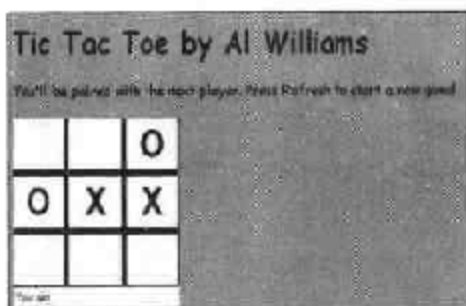


图 10.1 允许两人玩的浏览器到浏览器的通信

在网上有很多使用这种技术类型的应用例子，看看 play.yahoo.com，那里有一些有趣的多角色游戏，也可以在 station.sony.com 上找到 Jeopardy、Wheel of Fortune 以及其他更多这种类型的游戏。

当然，服务器的通信不仅仅在于玩网络游戏，服务器通信包括很多重要的应用，如客户

服务器，从读新闻组到聊天 applets，你在 chat.yahoo.com 上会发现使用 Java applet 的实时聊天，如果想使用 Java applet 来导航新闻组，提供这种服务的知名公司已经有很多了（我对什么使得.COM 从 2000 年的衰退中复苏不太了解）。使用这些相同的原则，可以很容易地想象基于 Java 的邮件客户端或者协作式的公告是怎么回事。

10.1.6.1 服务器协议

这个计划的目标是让两个相同的 applets 运行在不同的机器上，从表面来看，它们之间通信足够好，这样用户就可以互相玩 tic-tac-toe 游戏了。实际上，两个 applets 都与服务器通信，而不是直接通信。具体的协议必须定义好 applets 可以与服务器一起进行的事务。

为简化问题，我决定将服务器和客户端之间发送的数据限制为单字节命令。当第一个人连接时，服务器发送一个 W 字符通知他的 applet，游戏的启动还没有准备好，一旦第二个人连接上时，服务器送一个 X 给第一个人，送一个 O 给第二个人，这就将第一个人附值为 X，第二个人附值为 O。

可能最简单的命令就是从 0 到 8 的 ASCII 位，当 X 客户端接收到该命令时，它指出 O 客户端已经移动到指定的方块。当 X 客户端发送这个命令时，它告知服务器 X 作了一次移动。直到服务器发送一个 M 命令“邀请”移动，两个客户端才可以发送该命令。

在游戏的过程中，两人都可能退出游戏或者断连。发生这种事情时，服务器发出 Q 命令到两个客户端。客户端也可以发送一个 Q 来初始化断连。可能出现的其他的命令包括 W（win）、L（lose）和 G（game over）。

不像大多数服务器，它们每个套接字都使用单线程，而 tic-tac-toe 服务器为每一对套接字使用一个线程，一个线程处理两个角色是可能的，因为协议比较简单。Applet 客户端只有在服务器允许的时候才可以发送数据，这样单线程比较适合于每一对玩游戏的人。

10.1.6.2 服务器内幕

服务器惊人得简单（见清单 10.9）。主函数在 9001 端口上创建一个新的服务器，接着，它通过循环接受两个套接字的连接，并创建一个新的线程，将两个套接字通过构造函数传给线程对象。

清单 10.9 该服务器允许两个 applets 互相通信

```
// tic-tac-toe server

import java.net.*;
import java.io.*;

public class TicServ {
    // write and flush to a socket
    public static void swrite(Socket sock,int s) throws IOException {
        OutputStream str=sock.getOutputStream();
        str.write(s);
        str.flush();
    }
}
```

```

    }

    public static void main(String args[]) throws Exception {
        System.out.println("Tic Tac Toe Server by Al Williams");
        ServerSocket server = new ServerSocket(9001);
        // wait for connection pair and spin off thread
        while (true) {
            Socket sockx = server.accept(); // accept new connection
            write(sockx, 'W'); // wait      // tell player 1 to wait
            Socket socko = server.accept(); // accept player 2
            new BackgroundTask(sockx, socko).start(); // start new thread
        }
    }
}

class BackgroundTask extends Thread {
    private Socket sockx;
    private Socket socko;
    private OutputStream strx;
    private OutputStream stro;
    private InputStream istrx;
    private InputStream istro;
    private int [] game = new int[9];
    BackgroundTask(Socket sockx, Socket socko) throws IOException
    {
        // cache away sockets and create I/O streams
        this.sockx = sockx;
        this.socko = socko;
        strx = sockx.getOutputStream();
        istrx = sockx.getInputStream();
        stro = socko.getOutputStream();
        istro = socko.getInputStream();
    }

    // write to stream
    private void wflush(OutputStream os, int s) throws IOException {
        os.write(s);
        os.flush();
    }

    public void run() {
        int ct=0;
        int i;
        for (i=0; i<9; i++) game[i]=0;
        try {
            // set up both sides
            wflush(strx, 'X'); // X or O
            wflush(stro, 'O');
        }
    }
}

```

```

while (true) {
    int cmd;
    ct++; // next turn
    // if ct==10 then draw game
    if (ct==10) {
        wflush(strx, 'G');
        wflush(stro, 'G');
        break;
    }
    wflush(ct%2==0?stro:strx, 'M'); // ask player for move
    cmd= (ct%2==0?stro:istrx).read(); // wait for it
    if (cmd=='Q') break; // quit!
    wflush(ct%2==0?strx:stro, cmd); // pass to peer
    // store move so we can check for wins
    game[cmd-'0']=ct%2==0?'O':'X';
    // check for wins
    // X win?
    if ((game[0]=='X' && game[1]=='X' && game[2]=='X') ||
        (game[3]=='X' && game[4]=='X' && game[5]=='X') ||
        (game[6]=='X' && game[7]=='X' && game[8]=='X') ||
        (game[0]=='X' && game[3]=='X' && game[6]=='X') ||
        (game[1]=='X' && game[4]=='X' && game[7]=='X') ||
        (game[2]=='X' && game[5]=='X' && game[8]=='X') ||
        (game[0]=='X' && game[4]=='X' && game[8]=='X') ||
        (game[6]=='X' && game[4]=='X' && game[2]=='X')) {
        // X wins
        wflush(strx, 'W');
        wflush(stro, 'L');
        break;
    }
    // O win?
    if ((game[0]=='O' && game[1]=='O' && game[2]=='O') ||
        (game[3]=='O' && game[4]=='O' && game[5]=='O') ||
        (game[6]=='O' && game[7]=='O' && game[8]=='O') ||
        (game[0]=='O' && game[3]=='O' && game[6]=='O') ||
        (game[1]=='O' && game[4]=='O' && game[7]=='O') ||
        (game[2]=='O' && game[5]=='O' && game[8]=='O') ||
        (game[0]=='O' && game[4]=='O' && game[8]=='O') ||
        (game[6]=='O' && game[4]=='O' && game[2]=='O')) {
        // O wins
        wflush(stro, 'W');
        wflush(strx, 'L');
        break;
    }
}
} catch (Exception e) {
    e.printStackTrace(); // this shows up on server console
}

```



```

        // somebody quit, tell everyone
        // Each is in its own exception block so that if one throws an
        // exception, the other statements still execute
        try {
            wflush(strx, 'Q');
        } catch (Exception e) { };
        try {
            wflush(stro, 'Q');
        } catch (Exception e) { };
        // close sockets
        try {
            sockx.close();
        } catch (Exception e) { };
        try {
            socko.close();
        } catch (Exception e) {}
    }
}

```

所有的实际工作都在线程对象中产生 (BackgroundTask)。任务构造器为两个套接字设置输入输出流。而 run 方法主要控制游戏的逻辑。

Run 方法跟踪移动的序号有几方面原因, 第一, 奇移动相对于 O 角色, 而偶移动相对于 X 角色; 而且, 九次移动之后, 游戏也就结束了 (除非有人在这之前就已经胜出了)。

服务器最复杂的部分在于那种检查胜者的强制性逻辑。余下的代码简单地处理两个客户端之间的命令协议。

10.1.6.3 Applet 内幕

Applet 代码比服务器代码要稍微复杂一些 (见清单 10.10)。大量的代码总用于处理用户接口, 为简化问题, 我在一个网格面板上使用 9 个按钮形成 tic-tac-toe 面板, 没有在按钮之间画线, 只是将按钮下的面板设置成黑色, 同时通知网格面板对象在按钮之间使用大间隙, 这样不用太费力就可以取得 tic-tac-toe 的面板效果。

清单 10.10. 该 applet 用作 tic-tac-toe 服务器的用户接口

```

/// This is the tic-tac-toe applet
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

// Just a simple button with an integer tag
// could have used the ActionCommand, but that's a string
class TagBtn extends Button {
    public int tag;
}

```

```
TagBtn(String label, int tag) {
    super(label);
    this.tag=tag;
}
}

// The applet
public class TicTacToe extends Applet implements ActionListener {
    // arrays for buttons
    public TagBtn [][] btns = new TagBtn[3][3];
    // status message
    public Label lbl;
    // X or O
    public String player;
    // no moves allowed unless this is true
    public volatile boolean mymove=false;
    // my move
    volatile int move;
    int xsize=20; // size of X and O
    String server; // remote game host

    public void init() {
        int x;
        int y;
        String tmp;
        // Set X or O size
        tmp=getParameter("XOSize");
        try {
            xsize=Integer.parseInt(tmp);
            if (xsize==0) xsize=20;
        } catch (Exception e) { xsize=20; }
        // Get host name
        server=getParameter("Server");
        // set up GUI
        // The GUI has a border layout that contains a grid layout
        // The status window is south and the grid is north
        BorderLayout main = new BorderLayout();
        setLayout(main);
        Panel tictac = new Panel();
        tictac.setBackground(new Color(0,0,0));
        lbl = new Label("Tic Tac Toe by Al Williams");
        add(tictac, BorderLayout.CENTER);
        // gap of 5 makes the grid show up nicely
        GridLayout ly = new GridLayout(3,3,5,5);
        tictac.setLayout(ly);
        // Init buttons
        for (x=0;x<3;x++) {
```

```

        for (y=0;y<3;y++) {
            btns[x][y]=new TagBtn(" ",y*3+x+'0');
            btns[x][y].addActionListener(this);
            btns[x][y].setFont(new Font("Helvetica",Font.BOLD,xsize));
            tictac.add(btns[x][y]);
        }
    }
    add(lbl, BorderLayout.SOUTH);
    // start a thread to do all the network stuff
    new PlayThread(this).start();
}

// button press?
public synchronized void actionPerformed(ActionEvent e) {
    TagBtn b = (TagBtn)e.getSource();
    // no move if not your turn or button has X or O in it already
    if (!mymove || !b.getLabel().equals(" ")) return;
    move=b.tag; // tell the thread
    notify();
    b.setLabel(player);
}

// This routine gets the next move... it won't return
// until the actionPerformed function calls notify
public synchronized int getMove() throws InterruptedException {
    wait();
    return move;
}
}

// This is the network thread
class PlayThread extends Thread {
    private TicTacToe host;

    PlayThread(TicTacToe host) { this.host = host; }
    public void run() {
        int c;
        // connect to host
        Socket sock;
        InputStream in;
        OutputStream out;
        try {
            sock=new Socket(host.server,9001); // port # could be a param
            in = sock.getInputStream();
            out=sock.getOutputStream();
            // this loop waits for our partner to connect
            while (true) {

```

```
c=in.read();
if (c=='Q') {
    host.lbl.setText(
        "Other player quit. Press Refresh for a new game.");
    return;
}
if (c=='X' || c=='O') {
    host.player=c=='X'?"X":"O";
    // small flaw... player X hardly gets a
    // chance to see this message
    host.lbl.setText("You are " + host.player);
    break;
}
if (c=='W') host.lbl.setText("Waiting....");
}
// both players ready to go!
while (true) {
    c=in.read();
    if (c==-1) continue; // no input
    if (c=='G') {
        host.lbl.setText("Game over! Press Refresh.");
        return;
    }
    if (c=='W') {
        host.lbl.setText("You win! Press Refresh.");
        return;
    }
    if (c=='L') {
        host.lbl.setText("You lose! Press Refresh.");
        return;
    }
    if (c=='Q') {
        host.lbl.setText("Other player quit. Press Refresh.");
        return;
    }
    // My turn?
    if (c=='M') {
        host.mymove=true; // enable UI
        host.lbl.setText("Your turn");
        int m=host.getMove(); // wait for user to respond
        host.lbl.setText(""); // disable UI
        host.mymove=false;
        // send it
        out.write(m);
        continue;
    }
    // hmmm must be a move for opposite player
    c=c-'0';
}
```

```

        host.btns[c%3][c/3].setLabel(
            host.player.equals("X")?"O":"X");
    } // end while(true);

    } catch (Exception e) { e.printStackTrace(); }
    }
}

```

一旦 applet 初始化以后，它就启动一个线程，处理面向服务器的通信。线程的代码大部分都比较直观。它绝大部分时间用于等待来自服务器的命令，并相应地对用户接口进行更新。

从某种角度来看，代码变得有些复杂了。当线程收到来自服务器的一个 M 命令时，它会将 applet 中的 mymove 变量值设为 true。直到该变量的值设定以后，用户接口代码才会注意游戏中的所有移动操作。

一旦线程设置了这个变量，它必须等待，直到用户出现了移动操作。初始化移动的一种方法是使用一个循环简单地对 applet 查询是否有移动。这样效率不是很高，但是，服务器的线程在等待的过程中仍在继续执行。

初始化移动操作的一个更好的方法是使用 Java 的固有的线程同步特性。Applet 中的 getMove 方法调用 wait。这个调用会引起所有线程停止，直到 applet 对象的另一块代码调用 notify 方法。当一方移动时，applet 存储移动变量中的平方序号，并调用 notify。为了能够工作，每种方法必须在它们的声明中有 synchronized（同步）关键字。

10.1.6.4 综合

客户端都是 applet，所以需要有一个 HTML 页面来启动它们（见清单 10.11）。Applet 让你通过使用 xosize 参数来指定移动片数的大小。同时必须对运行游戏服务器的计算机命名，使用的是 server 参数。记住，服务器名必须与 applet 的逻辑位置相匹配，在玩游戏的人装载 HTML 页面之前必须运行 Web 主机上的服务器（也就是 Web 服务器）。

清单 10.11 该 HTML 页装载 tic-tac-toe 的 applet

```

<HTML>
<HEAD>
<TITLE>Tic Tac Toe!</TITLE>
</HEAD>
<BODY BGCOLOR=GRAY>
<H1>Tic-Tac-Toe by Al Williams</H1>
<P>You'll be paired with the next player. Press Refresh to start a new game!</P>
<APPLET code=TicTacToe.class HEIGHT=200 WIDTH=200>
<PARAM name="server" value="www.al-williams.com">
<PARAM name="xosize" value="40">
</APPLET>
</BODY>
</HTML>

```

测试整个系统的最简单的方法是将所有东西都放到一台计算机上。开始运行服务器，接

着可以使用 applet 浏览器装载 HTML 文件，在网页上设置服务器参数 localhost，例如，在 Windows 98 下，可以启动一个 MS-DOS 提示或者命令窗口，接着输入：

```
start java TicServ
start appletviewer tictac.html
start appletviewer tictac.html
```

如果你需要在外部的一台 Web 服务器上配置这个游戏，将需要某种访问，允许配置程序而不仅仅是数据。很多 Web 主机供应商不允许这种类型的访问，至少是在基本层次的服务上。因此你的网站上试图运行该程序之前要与你的供应商核实一下该问题。如果你的服务器未受防火墙或者网络地址转换器（就像很多 DSL 连接一样）保护，就必须打开 9001 端口，以使外界能够打开 PC 上的那个端口。

10.1.6.5 其他方法

你可以使用其他很多方法改进 tic-tac-toe 游戏，有些交互聊天的形式将会很适用，因此你可以用来奚落你的对手，如果用户可以选择昵称也很好，这两个方法都不是很难实现，但是它将让协议变得有些复杂。

10.2 快速解决方案

10.2.1 探寻 HTTP 协议规范

现在常用的 HTTP 协议有两个版本，1.0 版在 RFC1945 中有定义，就很多应用而言，该版本很有用，因为它比 1.1 版要简单一些，1.1 版增加了很多与虚拟主机、代理服务器以及缓存相关的特征。你可以在 RFC2616 中找到 1.1 版的规范。

10.2.2 创建简单请求

简单的请求主要是打开一个 TCP 端口，连向 Web 服务器（通常在 80 端口上），接着发送一行命令，请求一篇文档：

```
GET /index.html
```

对 0.9 版服务器进行合适的操作会简单的返回你要请求的文档。但是，有些服务器总会返回标题和文档，就像你做了一个新式的请求一样。

提示：因为服务器处理该类请求并不一定很可靠，除非有必要，你不应该使用简单的请求。产生 1.0 版的请求麻烦不了多少，但它会产生一致的响应。

10.2.3 创建 1.0 版的请求

1.0 版的请求与 0.9 版的请求表面上很类似，除了你可以包括定义有关请求信息的标题以外。像邮件消息一样，标题将以一个空行来结束。

另外，使用 1.0 版的请求中，不只有 GET 一个命令可以使用。RFC 定义了几个动词，但是最常见的是 GET 和 POST（使用 POST 发送数据到服务器）。

要创建一个请求，步骤如下：

1. 用动词、文档和你使用的 HTTP 版本号形成一行，发送请求。
2. 发送你要传给服务器的任何标题。
3. 发送一个空行。
4. 如果你在执行一个 POST 命令，可以发送更多数据。

下面是一个不带标题的请求的例子：

```
GET /test.htm HTTP/1.0
note that this line is blank
```

假设你要包括一个 User-Agent 标题来识别正在使用的客户端程序，那么请求将如下所示：

```
GET /test.htm HTTP/1.0
User-Agent: java Program by Al Williams
note that this line is blank
```

10.2.4 创建 1.1 版的请求

1.1 版的请求几乎与 1.0 版请求相同（当然，版本号不同），但是，所有的 1.1 版的请求必须至少有一个标题。这个标题是 host 标题，并且允许服务器支持更多主机作出合适响应。当然，你也可以添加其他标题。

1.1 版允许你使用 Connection 标题来维持一个到服务器的不变连接。当然，你如果保持套接字向服务器开放，将不能断定文档是否完整。所以，1.1 版服务器经常用一个大块编码来响应。如果你不想让套接字一直打开，可以发送 Connection: close 标题。但是，可能你还要处理大块编码（由 Transfer-Encoding 标题返回）。当大多数服务器使用永久连接时，只发送大块数据。

大块数据响应步骤如下：

1. 大块响应的第一个元素是 16 进制位的 ASCII 串，表示该块中数据的字节数。后边的字节将是响应文本部分。
2. 当一大块数据完成以后，另一块可以接着传输。
3. 最后一块数据长度为 0，并且没有文本。
4. 紧跟最后一块，有一个尾块，可能包含更多标题（决定于 Trailer 标题中列出的标题）。尾块的末尾是一个空行。

下面是来自使用大块数据响应方式的服务器的一个典型响应：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 19 Jun 2001 20:32:28 GMT
Content-Type: text/html
Cache-control: private
Transfer-Encoding: chunked
```

```
14
6/19/2001 3:32:28 PM
0
Note: this line is blank
```

10.2.5 读状态码

如果你创建了一个简单请求，服务器只是返回你请求的文档（或者是一篇错误文档）。没有简单的方法来确定那个请求是否会产生错误。但是，其他类型的请求返回一个响应码，如下所示：

```
HTTP/1.1 200 OK
```

这里，200 表示成功。你可以在表 10.1 中找到常用状态码的列表。

表 10.1 常用 HTTP 状态码

状态码	定义
100	服务器将接受请求（只用于 HTTP1.1）
200	成功
204	没有响应
301	持久移动
302	暂时移动
304	没有修改
400	最差的语法
401	未授权
403	禁止
404	未发现
500	内部错误
502	服务器太忙
503	网关超时

10.2.6 通过 HTML 向服务器发送表单数据

HTTP 的 POST 动作允许客户端发送数据到服务器，下面是一个简单的 HTML 表单：

```
<FORM ACTION=http://www.al-williams.com/aForm.jsp METHOD=POST>
Your name: <INPUT TYPE=TEXT NAME=myName><BR>
<INPUT TYPE=SUBMIT VALUE=Go>
</FORM>
```

当用户按下提交按钮时，产生的 HTTP 请求将如下所示：

```
POST /aForm.jsp HTTP/1.0
Content-Length: 24
Content-Type: application/x-www-form-urlencoded
```

```
myName=A1
```

其中 METHOD 属性可能有两种值，一个是例子中使用的 POST，将数据作为请求的正文发送，另一个值是 GET，它将数据放到 URL 上，这样一个 GET 请求可能如下所示：

```
GET /aForm.jsp?myName=A1 HTTP/1.0
```

10.2.7 使用 Java 发送表单数据到服务器

使用 Java 程序模仿表单提交并不很难，通过使用 GET 方法简单地创建一个 URL。使用 POST 有点困难，步骤如下：

1. 使用串创建请求的正文。
2. 使用独立的串来创建标题。
3. 通过正文串长来确定标题串中的 Content-Type 标题值。
4. 追加标题和正文串。
5. 向服务器写数据。

下面是执行这些步骤的一个代码片段：

```
String request;
String data;
// request data
data="tracknbr="+args[0]+"\\r\\n";
// Set up request
request="POST /cgi-bin/cttgate/ontrack.cgi HTTP/1.0\\r\\n";
request+="Content-Type: application/x-www-form-urlencoded\\r\\n";
request+="Content-Length: " + (data.length())+"\\r\\n\\r\\n";
request+=data;
// Write it out
str.write(request.getBytes()); // str is the socket's output stream
```

10.2.8 URL 数据编码

当发送数据到服务器时，必须对它进行编码，以隐藏一些特殊字符，如空格字符等。方法是将空格转换成“+”号，将其他可能被误解的字符转换成它的 16 进制值，前边带上“%”前缀。使用这种标记法，可以对整个串进行编码，因此串 PAT 可以换成%50%41%54。但是，由于这样会使串长变为原来的三倍，通常只对 URL 中有特殊意义的字符进行编码(例如，%，？，#)。

在 java.net 包中有一个 URLEncoder 类，可以完成上述工作：

```
public class URLEncode {
    public static void main(String args[]) {
        System.out.println(java.net.URLEncoder.encode(args[0]));
    }
}
```

```
    }
}
```

这里，静态的方法 `encode`（类中的唯一方法）将第一个参数转换成正确的格式，因此如果你输入：

```
java URLEncoder "A man & a plan = 100%"
```

将可以看到如下输出：

```
A+man+%26+a+plan+%3D+100%25
```

`&` 字符变成 `%26`，等号变成 `%3D`，“`%`”号变成 `%25`。尽管你可以对空格编码成 `%20`，使用“`+`”号作标记效率会更高（当然，那意味着“`+`”号要转换成 `%2B`）。

有一个相应的类来对这些串解码，将它们转换成它们的原始串。该类为 `java.net.URLDecoder`，静态方法是 `decode`。

10.2.9 自动提交表单

因为将数据提交给表单比较常见，将它包装到一个类里实现大部分工作就很容易了。清单 10.12 显示了这样一个类，对象从 `java.util.Properties` 那里继承而来。代码有一个 `main` 例子函数模仿清单 10.3 中所做的工作，也就是说，它发送一个 `tracknbr` 域到美国邮局服务的网站去跟踪一个数据包。

`main` 程序显示出发送数据很容易，对你想发送的每个域，只要在扩展的 `Properties` 对象里创建一个入口（使用相同的名字）。例如：

```
post.put("tracknbr", "03006000000373212757");
```

一旦设置好了所有的域，简单地调用 `send`，提供网页名以及与套接字输出相符的 `OutputStream` 即可。

清单 10.12 该类简单的向网页发送数据

```
import java.net.*;
import java.util.*;
import java.io.*;

// Make an HTTP 1.0 post
public class HTTPForm extends Properties {
    public boolean send(String page, OutputStream outstr)
        throws IOException {
        StringBuffer data=new StringBuffer();
        String headers;
        Enumeration i=keys();
        while (i.hasMoreElements()) {
            String key=(String)i.nextElement();
            data.append(URLEncoder.encode(key)
                + "=" +
                URLEncoder.encode((String)get(key)) +
                "&");
        }
    }
}
```



```

    }
    data.deleteCharAt(data.length()-1); // remove last &
    headers="POST " + page + " HTTP/1.0\r\n"+
        "Content-Type: application/x-www-form-urlencoded\r\n" +
        "Content-Length: " + data.length() + "\r\n\r\n";
    outstr.write(headers.getBytes());
    outstr.write(data.toString().getBytes());
    return true;
}

// Test main -- duplicates MailFind
public static void main(String args[]) throws Exception {
    HTTPForm test=new HTTPForm();
    Socket sock=new Socket("new.usps.com",80);
    OutputStream output=sock.getOutputStream();
    InputStream input=sock.getInputStream();
    test.put("tracknbr",args[0]);
    test.send("/cgi-bin/cttgate/ontrack.cgi",output);
// For this example, just dump results
    int c;
    do {
        c=input.read();
        if (c!=-1) System.out.print((char)c);
    } while (c!=-1);
    sock.close();
}
}

```

10.2.10 发送和接收 Cookies

Cookie 是 RFC2965 中定义的少量数据。其思想很简单：每个服务器可以要求浏览器存储那些会返回将来请求值的数据。不是所有的浏览器都支持 cookie，而且浏览器支持 cookie 的方式经常不同。

同时，浏览器（或者用户）可能会在任意时候选择性地删除或者中断块数据，因此不能使用 cookie 存储非常重要的数据。使用 cookie 最好的地方是用于维护状态。

通常，HTTP 事务没有什么状态，当你从相同的服务器那里请求两个页面时，你就要创建两个完整的独立事务，服务器没法知道哪两页要送到相同的客户端（除非，可能通过检查客户端的地址，那也不太可靠，因为很多用户使用了防火墙、网关或者 NAT 路由器（即网络地址转换路由器））。

服务器用于检查每个请求的一个常见的方案是使用如下方法：

1. 检查包含一个任意 session ID 的 cookie。
2. 如果这个 session ID 存在，就查询本地的 session 数据库。
3. 如果 cookie 为空，或者本地没有该 session ID，服务器就产生一个新 ID，并将它发给

浏览器。

客户端使用 Cookie 标题发送 cookie 到服务器，如果服务器需要浏览器来存储 cookie，它可以使用 Set-Cookie 来响应。

如果你使用高级的服务器工具，如 JSP，它们将为你产生处理 cookie 的逻辑。但是，如果你要写自己的 Java 客户端，可能需要存储 cookie，并将它们发送到服务器，以确保合适的操作。可以在“深入介绍”一节里找到使用 cookie 的例子（见清单 10.4 至 10.8）。

10.2.11 打开浏览器到浏览器的通信

Web 服务器影响 Web 浏览器的一个方式是通过装载和执行 Java applet。这些 Java 程序在浏览器上运行。但是，大多数浏览器不会让未知的 applet 做任何危及系统安全的事。这包括打开大多数网络连接的操作。

这种安全限制的一个例外是 applet 可以打开一个向它的源服务器的网络连接。通过在你的 Web 服务器机器上创建一个用户化的 Java 服务器，它有可能让 applet 通过服务器连向另一台 Web 计算机（例如，另一个 Web 浏览器）。

例如，在“深入介绍”一节中显示了一系列程序，允许你与另一个用户连接同一台服务器玩 tic-tac-toe 游戏。这需要你在相同的 Web 服务器上创建一个服务器程序。

通常，通过一个服务器连接，步骤如下：

1. 在接收端口上监听收到的来自 applet 的连接。
2. 允许 applet 连接另一台机器上的另一个端口。
3. 将远程机器的 applet 发过来的任何东西转发，反过来，将远程机器发过来的任何东西转发给 applet。

清单 10.9 中的 tic-tac-toe 服务器有点难。它的任务是将玩游戏的人配对，这样它实际上不用让 applet 指定它自己的连接，而是服务器将玩游戏的人配到某一对，但是，根本思想是相同的。

10.2.12 检查合法链接

扫描网页检查合法链接的 Java 程序显示了本章中涉及到的概念，该程序列在清单 10.13 中，该程序的执行步骤如下：

1. 程序创建一个 HTTP1.0 的请求，请求名为命令行上的串值的整个网页。
2. 接着程序将网页的文本选出来，找到 HREF 属性，程序并不试图排队那些注释或者脚本中的链接，这样它可能得到坏的链接，但也没什么坏处。
3. check 例程分离出 URL 部分，并解析相对链接，它接着发出一个 HTTP1.1 的 HEAD 请求。没有必要取回整个正文，因为唯一有意义的地方是响应码。
4. 如果服务器返回响应码 200，程序将指出链接合法，否则，它将报告该链接不合法（甚至重定向也被考虑成非法，因为你很可能直接链接到一个新位置）。

清单 10.13 该程序检查一个网页中的链接是否合法

```
// Link Checker
import java.net.*;
import java.io.*;
import java.util.*;

public class LinkCk {

    static int lastStatus=0;

    public static boolean check(String URL,String host,String root)
        throws IOException {
        String theHost=host;
        String dir="";
        int n,n1,port=80;
        // interpret URL
        n=root.lastIndexOf('/');
        if (n>0) dir=root.substring(0,n+1);
        if (dir.length()!=0 && dir.charAt(0)!='/') dir="/" + dir;
        // check for protocol string other than http:, forget it
        n=URL.indexOf(':');
        n1=URL.indexOf('/');
        if (n!=-1 && (n1>n || n1== -1)) {
            if (URL.substring(0,5).compareToIgnoreCase("http:")!=0)
                return true;
            n=URL.indexOf('/',7);
            if (n== -1) {
                theHost=URL;
                URL="/";
            } else {
                theHost=URL.substring(0,n);
                URL=URL.substring(n);
            }
            theHost=theHost.substring(7); // remove http://
        }
        else {
            if (URL.charAt(0)!='/') URL=dir+URL;
        }

        n=theHost.indexOf('@'); // userid/password?
        if (n!=-1) theHost=theHost.substring(n+1);
        n=theHost.lastIndexOf(':');
        try {
            if (n!=-1) {
                port=Integer.parseInt(theHost.substring(n+1));
                theHost=theHost.substring(0,n);
            }
        }
```

```

        // try to guess if this is a directory
        // this is an imperfect guess
        if (URL.indexOf('.')!=-1&&URL.charAt(URL.length()-1)!='/')
            URL=URL+"/";
        // make the request and see if it works
        String cmd="HEAD " + URL + " HTTP/1.1\r\n",line;
        Socket sock = request(theHost,port,cmd);
        BufferedReader rdr=new BufferedReader(
            new InputStreamReader(sock.getInputStream()));
        line=rdr.readLine();
        if (line==null) return false;
        return getStatus(line)==200;
    }
    catch (Exception e) {
        System.out.println(e);
        return false;
    }
}

// Pick out the status
public static int getStatus(String response) {
    StringTokenizer token = new StringTokenizer(response);
    token.nextToken();
    lastStatus=Integer.parseInt(token.nextToken());
    return lastStatus;
}

// Make a Version 1.1. request
public static Socket request(String host,int port,String cmd)
    throws UnknownHostException, IOException {
    Socket skt = new Socket(host,port);
    cmd+="host: " + host +"\r\nconnection: close\r\n\r\n";
    skt.getOutputStream().write(cmd.getBytes());
    return skt;
}

public static void main(String[] args) throws Exception {
    String line;
    // make a 1.0 request here so we avoid the chunked response encoding
    String cmd="GET /" + args[1]+" HTTP/1.0\r\n";
    Socket skt = request(args[0],80,cmd);

    BufferedReader rdr=new BufferedReader(

```

```

        new InputStreamReader(skt.getInputStream()));
boolean headers=true;
try {
    // check status code
    line=rdr.readLine();
    if (getStatus(line)!=200) {
        System.out.println(line);
        System.exit(0);
    }
}
do {
    line=rdr.readLine();
    if (line!=null) {
        if (line.equals("")) {
            headers=false;
            continue;
        }
        // skip headers
        if (headers) continue;
        String Uline = line.toUpperCase();
        int n=0,n1;
        // pick out hyperlinks
        // don't try to exclude those commented out
        do {
            n=Uline.indexOf("HREF=",n);
            if (n!=-1) {
                n+=5;
                if (line.charAt(n)=='"') {
                    n1=line.indexOf('"',++n);
                }
                else if (line.charAt(n)=='\\') {
                    n1=line.indexOf('\\',++n);
                }
                else {
                    int n2;
                    n1=line.indexOf(' ',n);
                    n2=line.indexOf('>',n);
                    if (n1!=-1||n2<n1) n1=n2;
                }
            }
            // detect empty tags (why do people do this?)
            if (n==n1) continue;
            String link=line.substring(n,n1);
            if (link.charAt(0)=='#') continue; // local
            System.out.print(link + "... ");
            System.out.println(
                check(link,
                    args[0],args[1])?"OK":("*BAD* " + lastStatus));
            n=n1;
        }
    }
}

```



```
        } while (r!=-1); // try another line
    }

    } while (line!=null);
}
catch (IOException e) {    }
}

}
```

第 11 章 协议操作者

11.1 深入介绍

现在学习数学不再像过去那样，我想这主要是因为有了计算机，当我还是小孩时，没有多少计算机，而且它们比较昂贵。现在，你可以随意的使用计算器，并得到它们（计算器）的帮助，还可以选择大型的科学计算机本解决复杂的计算问题。

在计算机出现之前，你必须真正理解数据如何能够完成实际的工作。的确，我们有滑动规则，但是读取一个滑动规则要求你了解对数，现在，很难看到有人可以用手算来算对数或者平方根。

没有人想从事计算机才就能够完成的劳苦的计算工作，但是，很多人没有料到的是对于科学家和工程来说，数据是开发有关事物工作的直觉知识，以提供数据答案的好方法。例如，考虑一下物理定律，即使你不记得准确的公式也没关系，重要的是要懂得大气体积、温度和压力之间的关系（如，体积在温度不变、压力减小的情况下增大）。

数据模型可以帮助你理解很多处理，那经常比绝对的数字结果要有价值得多。但是，计算机让直觉数学变成了不为人知的艺术了。网络与之类似，每次你使用一个工具来抽象网络时，你就远离了具体的细节，而那些细节有助于你理解到底发生了什么事情。

本书的前 10 章覆盖了 Java 网络编程的相对底层的应用，即使那样，这些技术并没要求你了解有关 IP 协议头以及源路由的细节内容。但是，Java 提供了一个更高层的接口来进行网络编程，可以常常（但不总是）用来获取快速结果。

更高级的接口为 Web 页面工作得很好，但是，对其他协议来说，编程并不是十分友好，因此你可能想继续使用 Socket 对象（像在前些章里用过的一样）。

11.1.1 URL 内幕

URL 类代表一种资源，使用统一资源定位格式，如 `http://www.coriolis.com`。通常，如果你愿意，可以使用 URL 来构造对象，尽管有大量的构造函数允许使用一个个分串来指定 URL，而不是用一个字符串来指定。

一旦有了 URL 对象，就可以通过几个不同的方法来获取 URL 对应的内容。例如，可以调用 `openStream` 来获取与文档相对应的 `InputStream`。也可以调用 `getContent`，它将返回一个依赖于文档的 MIME 类型的对象。

那意味着你使用几行代码就可以获取一个网页（见清单 11.1）。但是，你不能访问标题，

也不能发送数据到服务器，因为那需要一个 `URLConnection` 对象（讨论略）。

清单 11.1 读一个网页的简单方法

```
import java.net.*;
import java.io.*;

public class EZUrl {
    public static void main(String[] args) throws Exception {
        URL url = new URL(args[0]);
        InputStream html = url.openStream();
        int c;
        do {
            c=html.read();
            if (c!=-1) System.out.print((char)c);
        } while (c!=-1);
    }
}
```

使用 `URL`（更具体一些，是 `getContent`）的另一个问题是：表示数据的类都是由 Sun 公司指定的类，除非你提供自己的类。它们没有文档，你不能确信它们是否会继续存在。为了更明白，试着使用清单 11.7 中的代码（在“快速解决方案”一节中）从网上获取更多的 GIF 和 JPG 文件。你将看到 GIF 和 JPG 文件都是 `sun.awt.image.URLImageSource` 对象。通过 FTP 的 `URL` 获取的文本文件也被包装成一个 `sun.net.www.content.text.PlainTextInputStream` 类。

11.1.2 `URLConnection` 内幕

使用 `URL` 对象读一个 `URL` 是很容易的，但是，如果你想对 HTTP 事务作更多的控制呢？也许你想传输数据给服务器端的脚本或者想读标题，那么很可能想要使用 `URL` 对象的 `openConnection` 方法。这个函数返回一个 `URLConnection` 对象。如果 `URL` 真的使用了 HTTP 协议（就网页的情况而言，与 FTP 下载不同），对象会返回一个 `URLConnection` 的子类，即所谓的 `HttpURLConnection`。这个连接对象可以让你在提交 `URL` 请求之前设置标题和请求。

例如，假设你想向 Web 服务器上的一个表单提交一个请求，你可能使用清单 11.2 中的代码来完成该任务。那段代码像通常一样创建了 `URL` 对象（在这里，它打开 InterNIC 的 WHOIS 表单）。程序会将 `URLConnection` 对象置入专门的子类里（`HttpURLConnection` 对象）。

在程序建立连接之前，它调用 `setDoInput` 和 `setDoOutput` 来表示它想读和写的那个 `URL`。而方法 `setRequestMethod` 告诉 `URL` 使用 HTTP 的 Post 方法提交表单数据。最后，`setRequestProperty` 方法设置请求标题，表示服务器可以等待表单数据。

一旦程序实现了这些细节，就可以使用 `connect` 方法连接了。从这点来看，程序要求 `URL` 的 `OutputStream` 使用 `getOutputStream` 方法。在该流中执行写操作，会使数据从程序流向 Web 服务器。流向服务器的数据应该进行编码，那也是 `URLEncoder` 对象的静态方法 `encode` 的目的。它将空格替换成“+”号，将特殊字符替换成 16 进制的序列，正是 Web 服务器想要的那

种方式。

清单 11.2 使用 URLConnection 提交数据

```
//Sample Code to Submit a Form
import java.net.*;
import java.io.*;

class insearch
{
    static public void main(String [] argv) throws Exception
    {
        URL url=new URL("http://www.internic.net/cgi-bin/whois");
        HttpURLConnection conn=(HttpURLConnection)url.openConnection();
        int c;
        conn.setDoInput(true);
        conn.setDoOutput(true);
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-type",
            "application/x-www-form-urlencoded");
        conn.connect();
        PrintWriter pout = new
            PrintWriter( new
                OutputStreamWriter(conn.getOutputStream(),
                    "8859_1"), true );
        pout.print("whois_nic=" + URLEncoder.encode(argv[0]) +
            "&submit=Search&type=domain");
        pout.flush();
        // read results
        System.out.println(conn.getResponseMessage());
        InputStream is=conn.getInputStream();
        do
        {
            char x;
            c=is.read();
            x=(char)c;
            if (c!=-1) System.out.print(x);
        } while (c!=-1);
    }
}
```

URLConnection 类有很多方法，其中一些并不是很有用，下面是最重要的几个方法：

- **getDefaultAllowUserInteraction**——该静态方法返回 true 或 false，这依赖于系统内部标志（用于确定对象是否可以提示用户输入数据，像输入密码那样）的缺省值。缺省值应用于后边所创建的实例。
- **setDefaultAllowUserInteraction**——该静态方法为所有新的实例设置交互标志。

- `getAllowUserInteraction`——返回该对象的交互标志值。
- `setAllowUserInteraction`——设置该对象的这个实例的交互标志。
- `getFileNameMap`——可以使用这个方法检索一个 `FileNameMap` 参考，可以使用该方法来猜测一个文件名的 MIME 类型（例如，.txt 文件将是属于 `text/plain` 类型）。
- `setFileNameMap`——如果想设置自己的 `FileNameMap` 对象，可以使用该方法。
- `guessContentTypeFromName`——该方法猜测一个文件名的 MIME 类型，它实际上调用缺省的 `FileNameMap` 对象，或者调用 `setFileNameMap` 来完成该任务。
- `guessContentTypeFromInputStream`——不使用 MIME 类型，你可以调用该方法来检查输入流的若干字节，来确定数据的类型。
- `getContent`——该方法有两个变体，第一个变体返回一个对象，该对象依赖于文档的 MIME 类型。例如，如果是图像文档，函数返回 `sun.awt.image.URLImageSource` 对象，如果是文本类型，则返回 `InputStream` 类型。这与本章前边提到的 `URL.getContent` 方法类似。第二个变体需要一组 `Class` 对象，该方法尽可能的返回其中的一种类型。
- `getContentLength`，`getContentEncoding`，`getContentType`，`getDate`，`getExpiration`，`getLastModified`——这些方法都返回与文档标题相应的值，如果可行的话。
- `setIfModifiedSince`——该方法设置 `IfModifiedSince` 标题。
- `getHeaderField`，`getHeaderFieldDate`，`getHeaderFieldInt`，`getHeaderFieldKey`——可以使用这些方法来获取任意的标题，其类型可以是 `String` 或者指定的数据类型（例如，`Date` 或者 `int`）。
- `setRequestProperty`——可以使用该调用来设置任意的标题。
- `setDoInput`，`setDoOutput`，`getDoInput`，`getDoOutput`——缺省情况下，对象将只是处理来自服务器的数据，但是，你可以使用这些方法来控制数据流的方向。
- `connect` 该方法实际上与服务器相联系，在调用 `connect` 之前，你必须设置好选项和标题。
- `getInputStream`，`getOutputStream` 这些方法返回可以用于与服务器通信的流。记住这些流要受 `setDoInput` 方法和 `setDoOutput` 方法的状态的支配。

可以看出，虽然 `URLConnection` 通常认为是用于一般的应用，它还是用来完成实际工作的操作者，如果你喜欢，你甚至可以创建自己的类。

11.1.3 URLConnection 子类

可以使用 `URLConnection` 的子类——`HttpURLConnection`（它是一个抽象子类）。在清单 11.2 中应用了这个类。获得该对象的唯一方法是使用 `http://URL` 创建一个 `URL` 对象，并调用对象的 `openConnection` 方法。接着可以将返回的 `URLConnection` 值赋给一个 `HttpURLConnection`。

这个特殊的子类允许使用 `setRequestMethod` 来设置请求类型（例如，GET 或者 POST），

使用 `getResponseCode` 来获得响应码，并执行其他的具体的 HTTP 任务。

除了具体的 HTTP 子类，针对 JAR (Java 压缩文档) 文件也可以使用特殊的子类，它们与你使用的 Java 类包的 JAR 文件相同。你可以从本地文件或者通过 HTTP 获得 JAR 文件。在 `nettest.jar` 文件里要为 `inside.htm` 文件创建一个 JAR 连接，也要创建一个 URL 对象。但是，在这种情况下，你提供了一个伪 URL，就像下边三个 URL 一样：

```
jar:file://c%3A/lib/nettest.jar!/inside.htm
jar:http://www.coriolis.com/jars/nettest.jar!/inside.htm
jar:ftp://ftp.coriolis.com/jars/nettest.jar!/inside.htm
```

在第一个例子里，JAR 文件是 `c:\lib` 目录下名为 `nettest.jar` 的本地文件，注意需要对冒号进行编码，但是 `“//”` 不需要编码。在第二个以及第三个例子里，JAR 文件定位在指定 URL 的一个服务器上。所有情况下，你都从 JAR 里获取 `inside.htm` 文件。感叹号只是用来终止伪 URL，并不是 JAR 文件名的一部分。感叹号后边的部分是你想要从 JAR 文件中获取的文件名。

一旦你使用特殊的 URL 来构造 URL 对象，就可从文件中读数据，就像你使用其他 URL 一样。那意味着你可以调用 `openStream` 或者获取一个 `URLConnection` 对象，并将它附给 `JarURLConnection`。

11.1.4 协议和内容操作者

URL 和 `URLConnection` 对象依赖于 `URLStreamHandler` 和 `URLConnection` 类来执行具体的协议处理。另外，一个 `ContentHandler` 类知道如何将收到的数据转换成 Java 类型。但是，官方的 Java 库没有任何这些类的实现，当你使用普通协议像 HTTP 或者 FTP 的时候，你使用的类实际上是 `sun` 包里边的类，而不是 Java 中的基础类。

11.1.4.1 客户协议

在经过封装后，URL 对象检查它的 URL 的协议端口，并调用实现了 `URLStreamHandlerFactory` 的一个对象（使用静态的 `URL.setURLStreamHandlerFactory` 方法设置该对象）。这个对象主要用于创建一个 `URLStreamHandler` 子类，使其与指定的协议相符。但是，你可以只安装一个 `URLStreamHandlerFactory`，一旦它设置好，就不能再改变。

子类 `URLStreamHandler` 创建一个相应的 `URLConnection` 对象（或者是 `URLConnection` 的子类）。它也解析 URL，这样你可以定义用户化 URL 格式（像 `JarURLConnection` 对象使用的 `jar:protocol` 格式）。对象 `URLConnection` 实际上与服务器进行通信。

通常，你不用担心这里面的任何环节，因为它的工作是透明的。但是，如果你想添加一个用户化的操作者，能通过创建自己的对象来实现。为何不去创建一个操作者呢？比如，你可能想扩展 Java 的 URL 对象，使它能知道 Finger 协议。你可以定义客户 URL，指向一个数据库表或者其他用户资源。首先你将不得不做一点工作，但是在你可以获得那个类之后，对于你来说（或者任何其他入），访问你的客户内容将很容易。你甚至可以在运行时装载新的操作者。

清单 11.3 显示了一个 `URLConnection` 的子类，它知道如何在 13 端口上打开一个时间服务器。绝大部分工作要使用 `connect` 方法。这个方法在正确的端口上打开套接字，并设置连接标志（`URLConnection` 的一部分）。因为这个连接直接隐藏在一个套接字里，`getInputStream` 方法是很普通的。

清单 11.3 这个特殊化的 `URLConnection` 类知道如何查询一个时间服务器

```
import java.net.*;
import java.io.*;

public class TimeURLConnection extends URLConnection {
    private Socket conn;
    public final static int DEFPORT=13;

    public TimeURLConnection(URL url) { super(url); }

    public synchronized void connect() throws IOException {
        if (!connected) {
            int port=url.getPort();
            if (port<=0) port=DEFPORT;
            conn=new Socket(url.getHost(),port);
            connected=true;
        }
    }

    public String getContentType() {
        return "text/plain";
    }

    public synchronized InputStream getInputStream() throws IOException {
        connect();
        return conn.getInputStream();
    }
}
```

URL 对象如何知道使用这个特殊的连接对象呢？需要的第一部分是一个 `URLStreamHandler` 子类，该对象知道如何创建正确的 `URLConnection` 对象，并且相当容易（见清单 11.4）。

清单 11.4 这个简单类代表 URL 类创建正确的 `URLConnection` 子类

```
import java.net.*;
import java.io.*;

public class TimeHandler extends URLStreamHandler {
    public int getDefaultPort() {
        return TimeURLConnection.DEFPORT;
    }
}
```

```

        protected URLConnection openConnection(URL url) throws IOException {
            return new TimeURLConnection(url);
        }
    }
}

```

你还必须要通知系统使用这个 `URLConnectionHandler` 对象，当你创建一个 `URLConnection` 对象时，Java 首先要看你是否安装了一个 `URLConnectionHandlerFactory` 类。如果已安装了，它就调用该类的 `createURLConnectionHandler` 方法。如果你不提供这个类（或者方法返回值为 `null`），系统将查看名为 `java.protocol.handler.pkgs` 的系统属性。该属性可能包含一个包名列表，由“`|`”符分开。

如果该属性有一个值，Java 就查找一个类与你试图使用的协议相匹配。例如，假设属性是“`com.al_williams.protocol.com.coriolis.java.phandlers`”，并且你试图装载一个 `mailto` 的 `URLConnection`，Java 将沿着 `CLASSPATH` 查找这些类：

```

com.al_williams.proto.mailto.Handler
com.coriclis.java.phandlers.mailto.Handler

```

如果这个类存在，它就是 `URLConnectionHandler` 的子类（和清单 11.4 中一样）。如果没有一个类与该描述相符，最后 Java 将查找一个名为 `sun.net.www.protocol.mailto.Handler` 的类（对 `mailto` 协议而言）。

当然，定制的 `URLConnectionHandlerFactory` 提供了对处理的最大程序的控制，同时也易于编写（见清单 11.5）。只要记住你可以只安装一个 `factory` 类，后边试图设置 `factory` 类都会抛出一个异常：

清单 11.5 `URLConnectionHandlerFactory` 类选择某协议的客户操作者

```

import java.net.*;

public class MyStreamHandlerFactory implements URLStreamHandlerFactory {
    public URLStreamHandler createURLStreamHandler(String protocol) {
        if (protocol.equalsIgnoreCase("time"))
            return new TimeHandler();
        return null; // huh?
    }
}

```

如果你正在使用定制的 `factory` 类，应该让它使用静态的 `URLConnection.setURLConnectionHandlerFactory` 方法，像清单 11.6 中的程序那样。

清单 11.6 这个简单程序使用客户协议操作者

```

import java.net.*;
import java.io.*;

public class TimeURLTest {
    public static void main(String [] args) throws Exception {
        URL.setURLConnectionHandlerFactory(new MyStreamHandlerFactory());
        URL url=new URL("time://tock.usno.navy.mil");
    }
}

```

```
InputStream is=url.openStream();
int c;
do {
    c=is.read();
    if (c!=-1) System.out.print((char)c);
} while (c!=-1);
is.close();
}
```

11.1.4.2 客户内容操作者

协议操作者和连接对象只与数据传输相关。但是，当你调用 `getContent` 方法时，URL 对象也将数据转换成 Java 对象。Java 如何知道使用什么类型的对象呢？那就是从 `ContentHandler` 那里继承的内容操作者的范畴了。

`getContent` 类本身实现了方法，返回合适的类型值。另有一种方法，调用程序可以指定一个类的列表，如果可能，`getContent` 将返回该列表中的一种类型值。

Java 用来查找合适的内容操作者的处理与用来查找协议操作者的处理很类似。首先，URL 对象调用 `URLConnection` 对象的 `getContent` 方法。如果你在写一个客户 `URLConnection` 对象，从这方面来看你可能要处理所有的事务。

对 `getContent` 的缺省处理，要看你是否使用 `URLConnection` 类静态的 `setContentHandlerFactory` 方法安装了 `ContentHandlerFactory` 类。如果安装了，Java 将调用 `createContentHandler` 方法查找内容操作者，如果该类没有安装或者 `setContentHandlerFactory` 方法返回值为 `null`，Java 将在 `java.content.handler.pkgs` 的系统属性检查包名。

`java.content.handler.pkgs` 属性包含一个包名的列表（由“|”符分隔开）。Java 使用 MIME 类型作为类名，并查找那个类的包列表。例如，为了确定 `application/x-video` 类型的操作者，Java 要检查类名为 `application.x_video` 的包（将“-”转换为“_”，因为“-”在类名中不合法）。

如果所有其他方法都失败，Java 将查找缺省的操作者，例如，对 MIME 类型来说，缺省的操作者将是 `sun.net.www.content.application.x_video`，在本章的快速解决方案一节中可以找到一个内容操作者的例子。

11.2 快速解决方案

11.2.1 获取 URL 的数据

如果你想得到与 URL 相对应的 `InputStream`，使用 URL 对象可以很容易地得到它。其使用步骤如下：

1. 创建一个 URL 对象，传一个代表 URL 值的字符串给它的构造函数。
2. 调用 URL 对象的 `openStream` 方法来获取 `InputStream`。

下面是一个代码示例（完整的清单见清单 11.1）：

```
URL url = new URL(args[0]);
InputStream html = url.openStream();
```

11.2.2 获取 URL 的内容

对很多 MIME 类型来说，你可以让 URL 对象直接返回一个与文档数据类型相对应的对象。对一个 HTML 文档，对应的对象应该是 InputStream 类型。但是，其他类型，例如像 image 类型，可以返回易用的 image 对象。

关键是调用 URL.getContent 方法。该方法检索数据，检查文档的 MIME 类型，并创建合适的对象，在清单 11.7 的程序里试着在命令行里输入不同的文档类型的 URL。你将看到由 getContent 返回的对象类型。

清单 11.7 该程序显示由 getContent 返回的对象类型

```
import java.net.*;
import java.io.*;

public class E2Url12 {
    public static void main(String[] args) throws Exception {
        URL url = new URL(args[0]);
        Object doc = url.getContent();
        System.out.println(doc.getClass().getName());
    }
}
```

11.2.3 设置请求标题

如果你想设置请求标题，就不能使用 URL 对象的简单方法。相反，你要使用 openConnection 方法来获取对象。

该对象提供了与服务器进行双向通信的功能。URLConnection 对象有若干方法允许你设置请求标题：

- setIfModifiedSince——允许设置 setIfModifiedSince 标题。
- setRequestProperty——设置任意的请求标题。
- setDefaultRequestProperty——设置缺省的标题（该方法是静态方法）。

缺省情况下，URLConnection 类只允许数据流入。但是，你也可以往服务器写数据，如果你调用 setDoOutput (true)。你也可使用 getInputStream 和 getOutputStream 来查找将用于执行输入输出的实际的流。

注意所有提到的方法（除了 getInputStream 和 getOutputStream 之外必须在连接服务器之前要调用以外（使用 connect））。

11.2.4 读取响应标题

`URLConnection` 对象也允许你读响应标题。你可以使用 `getHeaderField` 方法以 `String` 的形式来读取任意的标题，如果标题有特殊的格式，你可以使用 `getHeaderFieldDate` 或者 `getHeaderFieldInt` 将数据分别格式化成 `Date` 对象或者一个 `int`。

除了这些常用方法以外，对象也提供了方便的方法来获得具体的标题，例如 `getContentEncoding`, `getContentType`, `getDate`, `getExpiration` 和 `getLastModified` 方法。它们获取方法名指定的标题。

11.2.5 使用特定的 HTTP 连接

如果你向 `URL` 对象传递的 `URL` 以 `http` 开头，方法将返回的一个子类 `URLConnection`，特别地，你将接收到一个 `HttpURLConnection`。该类拥有普通的 `URLConnection` 类的所有方法，它们都是针对具体的 HTTP 连接的。在该类中你将发现下述方法：

- `setFollowRedirects`——允许你指定对象应该透明地处理重定向。
- `setRequestMethod`——设置 HTTP 方法（例如，`GET` 或者 `POST`）。
- `getResponseCode`——从服务器那里返回若干响应码。
- `getResponseMessage`——以 `String` 的形式返回响应消息。

当然，`openConnection` 方法总是返回一个 `URLConnection`。为了利用 `HttpURLConnection` 中的特殊类，你必须将返回值转换成正确的类型：

```
HttpURLConnection conn=(HttpURLConnection)url.openConnection();
```

11.2.6 传送数据到服务器

使用 `URLConnection` 对象，可以很容易地将数据传给 Web 服务器，要实现这个任务，必须要做以下几步：

1. 调用 `setDoOutput` 传递一个 `true` 参数。
2. 使用 `setRequestMethod` 来为 `POST` 设置请求方法。
3. 设置任何需要的请求标题。
4. 调用 `connect` 来初始化与服务器之间的通信。
5. 从 `URLConnection` 对象那里获取输入输出流。
6. 使用 `URLEncoder.encode` 方法对想要传递的数据编码。
7. 向输出流里写数据。
8. 从输入流里读结果。

你可以从清单 11.2 中的具体实现中找到相关代码。

11.2.7 打开一个 JAR 文件作为 URL

Java 定义了一个客户 URL 类型，允许你在本地文件系统或者通过网络操作 JAR 文件。对应这种类型 URL 的协议开始为 jar:specifier。该客户 URL 实际上包含另一个确定 JAR 文件放置位置的 URL。这个 URL 以感叹号结束。在感叹号后边，你可以放置你要访问的 JAR 中的文件名。

例如，假设你有一个 JAR 文件可以通过名为 nettest.jar 的 FTP 站点获得，需要在 JAR 范围内访问各类 jbb.txt 文件，你使用的伪 URL 将可能是：

```
jar:ftp://ftp.coriolis.com/jars/nettest.jar!/jbb.txt
```

注意：如果你想使用本地文件，可以使用文件协议。但是，在 Windows 或者 MS-DOS 下，你必须小心对路径名前的冒号编码为 %3A，但是，不要对 "/" 或者 "\" 进行编码，因为它们都是 URL 的合法部分，c:\binks.jar 对应的 URL 例子将是：jar:file://c:%3A/binks.jar!script.txt。

一旦你有特殊的 URL 格式，就可以简单地将它传给 URL 对象的构造函数，你可以把它看作是任意其他的 URL 资源。如果需要更多控制，可以调用 URL.openConnection，并将返回的 URLConnection 对象转换成 JarURLConnection 对象。该对象允许你包括 JAR 文件的说明和安全信息。

清单 11.8 显示了使用 URL 对象读一个 JAR 文件的程序。该程序带有两个参数：JAR 文件名以及 JAR 文件内部的文件名。

清单 11.8 该程序从 JAR 文件中读取文件名

```
import java.net.*;
import java.io.*;

public class JarPrinter {
    static public void main(String [] args) throws Exception {
        String jarfile;
        // MSDOS/Windows names have colons in them that need to convert to
        // %3A, but can't convert / or \ to their URL equivalents
        int n=args[0].indexOf(':');
        if (n!=-1)
            jarfile=args[0];
        else
            jarfile=args[0].substring(0,n)+"%3A"+args[0].substring(n+1);
        String urlstring = "jar:file://" + jarfile + "!/" + args[1];
        System.out.println("Opening: " + urlstring);
        URL url = new URL(urlstring);
        InputStream is=url.openStream();
        int c;
        do {
            c=is.read();
        }
    }
}
```

```

        if (c!--1) System.out.print((char)c);
    } while (c!--1);
    is.close();
}
}

```

11.2.8 创建一个客户协议操作者

Java 知道一些协议，例如 http、ftp 以及 JAR 文件伪协议。但是，你可以扩展系统，让它知道你自己的定制（客户）协议。你必须创建两个类来控制 URL 对象如何装载数据：

1. 大多数工作发生在从 URLConnection 继承而来的一个类，该类管理到 URL 文档的物理连接（典型地通过套接字）。
2. 同时要创建从 URLStreamHandler 继承的一个类。该类简单地定义了特定协议与你的 URLConnection 类之间的通信。

清单 11.9 显示了处理所谓 string 协议的一个 URLConnection 类。目的是使用该协议来代替 HTTP 协议，获取一个网页，以 Java 串的形式返回。

这个客户类简单地将传给 URL 构造函数的串里去掉协议标识，接着加上 http:// 协议标识符，最后，你将看到它将处理如下 URL：

```
string://www.coriolis.com
```

但是，该类并不处理内容到 String 的转换，在下例中你可以简单地重载该类的 getContent 方法，作任何必需的处理。但是，对于更复杂的系统而言，你将要在一个独立的 ContentHandler 对象里处理内容的产生。

清单 11.9 URLConnection 类使用 HTTP 协议载入一个页面

```

import java.net.*;
import java.io.*;

public class StringURLConnection extends URLConnection {
    private URL base;
    public final static int DEFPORT=80;
    public StringURLConnection(URL url) { super(url); }

    public String getContentType() { return "application/x-jstring"; }

    public synchronized void connect() throws IOException {
        if (!connected) {
            // remove old protocol and substitute http://
            String newurl="http://" + url.toExternalForm().substring(9);
            base=new URL(newurl);
        }
    }

    public synchronized InputStream getInputStream() throws IOException {

```

```
        connect();  
        return base.openStream();  
    }  
}
```

你可能认为该类满足你的所有要求，但是 Java 要求你还要定义一个类，从 `URLConnectionHandler` 继承而来。该对象的唯一目的是在需要时创建一个客户化的 `URLConnection` 对象的实例。清单 11.10 为清单 11.9 中的客户操作者完成这项任务。

清单 11.10 该简单类在需要时创建一个客户 `URLConnection` 对象的实例

```
import java.net.*;  
import java.io.*;  
  
public class StringURLHandler extends URLStreamHandler {  
    public int getDefaultPort() { return StringURLConnection.DEFPORT; }  
    protected URLConnection openConnection(URL url) throws IOException {  
        return new StringURLConnection(url);  
    }  
}
```

11.2.9 安装一个客户协议操作者

一旦写好一个客户协议操作者，就必须告诉 Java 当它在 URL 对象里见到合适的协议时要使用它。有两种方法可以完成这个操作：

1. 创建一个 `URLConnectionHandlerFactory` 类，并使用 `URLConnection.setURLConnectionHandlerFactory` 来注册它。客户类在一个协议串的基础上创建 `URLConnectionHandler` 对象。
2. 在一个独立的包中创建 `URLConnectionHandler` 对象，要使用这个方法，你必须恰当的设置专门的系统属性。

当使用第一个方法时，你可以对流操作者进行完全的控制。如果从 `createURLConnectionHandler` 方法那里返回 `null`，Java 会查找使用第二个方法的任何操作者。如果失败，Java 会查找它的缺省操作者。

注意：一旦你在程序里调用了一次 `setURLConnectionHandlerFactory` 方法，再次调用将会出错。

如果你选择使用第二个方法，必须将你的 `URLConnectionHandler` 类放到一个包里，协议名将充当子包名，类名必须叫做 `Handler`。例如，要处理 FTP，你可能要写一个 `com.coriolis.protohandlers.ftp.Handler` 类。另外，系统属性 `java.protocol.handler.pkgs` 必须包含包名（在这里是 `com.coriolis.protohandlers`），属性可以包含多个包名；可以使用“|”来将多个包名分隔开。

要为 URL 串（见上一节）安装操作者，你可以像清单 11.11 一样使用 `URLConnectionHandlerFactory`。

清单 11.11 URLStreamHandlerFactory 类创建一个客户流操作者

```
import java.net.*;

public class StringSHFactory implements URLStreamHandlerFactory {
    public URLStreamHandler createURLStreamHandler(String proto) {
        if (proto.equalsIgnoreCase("string"))
            return new StringURLHandler();
        return null;
    }
}
```

11.2.10 创建一个客户内容操作者

如果你要读一个客户协议，不只要将 `URLConnection` 类客户化，还要提供一个特殊的内容操作者来把你的数据编码成合适的 Java 对象。

完成这个任务的最简单的方法是重载 `URLConnection` 类的子类的两个版本的 `getContent` 方法。但是，这样就不够灵活，因为一个 `URLConnection` 可能要为不同类型的数据服务（例如，HTTP 连接可能要为文本或者图像服务）。

针对上述情形，你需要在 `ContentHandler` 类的基础上写一个客户内容操作者，该类将为 `URLConnection` 里的两个 `getContent` 方法提供思路。`getContent` 的第一个版本简单地返回缺省的数据类型，第二个版本接受一组对 `Class` 的引用，这个数组是调用者愿意接收的一个对象列表。`getContent` 方法将返回它能识别出的第一个对象类型。

当然，根据 `getContent` 的第二个版本去写它的第一个版本是很容易的，因此写第一个版本没有太多的工作要做，清单 11.12 显示了前一节中开发的 URL 串的一个实现。这个内容操作者将会满足一个 `InputStream`（像普通的 HTTP 请求一样）或者作为一个 `String` 的请求。

第一个版本的 `getContent` 使用一个缺省的数据类型数组（在这里，它里边包含有串）简单地调用第二个版本。为提高效率，当类返回一个串时，它从块中读数据并使用一个 `StringBuffer` 作为中间的工作区。

注意清单 11.12 中的类都位于一个包内（`com.al_williams.contenthandlers.application`）。那是因为下一个会话将安装这个操作者，这样 Java 会自动找到它。就像你在那一节里看到的一样，类名在这里是很重要的。

清单 11.12 这个操作者将一个网页读进一个串，并将其返回

```
package com.al_williams.contenthandlers.application;

import java.io.*;
import java.net.*;

public class x_jstring extends ContentHandler {
    public Object getContent(URLConnection urlc) throws IOException {
```



```

    Class [] clist=new Class[1];
    clist[0]=String.class;
    return getContent(urlc,clist);
}
public Object getContent(URLConnection urlc, Class [] clist)
    throws IOException {
    for (int i=0;i<clist.length;i++) {
        if (clist[i]==InputStream.class)
            return urlc.getInputStream();
        if (clist[i]==String.class) {
            StringBuffer rv=new StringBuffer();
            InputStream is=urlc.getInputStream();
            byte [] buffer=new byte[256];
            int len;
            do {
                len=is.read(buffer,0,buffer.length);
                if (len!=0)
                    rv.append(new String(buffer));
            } while (len==buffer.length);
            return rv.toString();
        }
    }
    return null;
}
}

```

11.2.11 安装一个客户内容操作者

一旦有了客户内容操作者，就必须告诉 Java 应该使用它。这与你安装一个协议操作者的方式很相似。接着，你有两个选择：

1. 创建一个 `ContentHandlerFactory` 类并使用 `URLConnection.setContentHandlerFactory` 注册。客户类在数据的 MIME 类型基础上创建 `ContentHandler` 对象。
2. 在单独的包里创建 `ContentHandler` 对象。为使用该方法，你必须恰当地设置一个专门的系统属性。

如果你使用第一个 `createContentHandler` 方法，就要写一个方法来创建正确的 `ContentHandler`。如果你不想处理特殊类型，可以返回 `null`，Java 将继续查找另一个操作者。

使用第二个方法要求你将你的内容操作类放到一个包里，如果 MIME 类型是 `application/x-example`，你就要将你的类命名为 `application.x_example`（注意“-”变为“_”）。同时必须设置 `java.content.handler.pkgs` 系统属性来指定该包。系统属性列表可以不止一项，可以使用“|”将多项分隔开。

注意清单 11.13 中的程序读现在的属性，并将客户包加到列表中。这会防止程序关闭终端用户已经安装的所有客户操作者。该包名为 `com.al_williams.contenthandlers`，指定的类是 `com.al_williams.contenthandlers.application.x_jstring`，它与清单 11.12 相对应。

清单 11.13 这个主程序使用系统属性安装一个内容操作者，并安装一个客户连接操作者工厂

```
import java.net.*;
import java.io.*;

public class URLStringTest {
    public static void main(String[] args) throws Exception {
// Custom Stream Handler Factory
        URL.setURLStreamHandlerFactory(new StringSHFactory());
        // Add our custom handler
        String path=System.getProperty("java.content.handler.pkgs","");
        if (!path.equals("")) path+="|";
        path+="com.al_williams.contenthandlers";
        System.setProperty("java.content.handler.pkgs",path);

        URL url=new URL("string://www.coriolis.com");
        Class [] clist = new Class[1];
        clist[0]=String.class;
        String s=(String)url.getContent(clist);
        System.out.println(s);
    }
}
```

第 12 章 解释 HTML

12.1 深入介绍

像很多作者一样，我偶尔也做做演讲，并讲讲课。我了解到的一件事就是当与翻译人员合作讲课时，只对翻译人员讲话是极不礼貌的。对外语翻译人员和手语翻译人员来说尤其如此。你演讲的对象是学生，不是翻译人员。

在前一章，程序大多数通过网络对数据进行读写处理。但是，很多程序的一个重要部分是要了解它们获取的数据。因为你要处理的大多数数据都是 HTML（超文本标识语言），研究 HTML 的实际结构同时也研究解析它的技术，显得比较有意义。

当然，解释 HTML 对不同的人有不同的含义。对最简单的终端，你可以只将 HTML 显示成浏览器所显示的样子。稍微有些挑战性的程序可能要抽取基本的文本。更漂亮的程序可能需要实现下述功能：能够理解文档，检查语法的合法性、检查链接，并以某些方式转换文档。

12.1.1 显示

很少有人需要写一个完全的 HTML 翻译程序，那是一件好事，因为 HTML 规范很复杂（并且随着它的版本修订变得越来越复杂）。表格、样式表和框架实现起来并不简单。

```
JLabel lbl = new JLabel("<HTML>I'm feeling <B>Bold</B></HTML>");
```

你不必以</HTML>标志结束，虽然这是一个好方法。有些 Java 的旧版本不喜欢上下文中大写的标签名，但是最近的 Java 版本处理它，没有问题。

但是，如果你想完全翻译 HTML，那就需要使用 JEditorPane。这是一个标准的 Swing 组件，你可以设置它，不允许它被编辑（因为你不想用户在网页上执行常规的文本编辑）。

清单 12.1 显示了一个非常简单的程序，将一些 HTML 页面载入一个 JEditorPane 组件。文本很难将它编码到程序里，但是你可以使用前两章中的任何方法从网上载入页面。

即使你不熟悉 Swing，可能只是对程序的最后部分感到困惑。JScrollPane 组件为 HTML 的显示增加了滚动条，并且 JFrame 对象在单个顶级窗口里包含了所有的东西。

清单 12.1 JEditorPane 组件能够显示 HTML

```
import javax.swing.*;

public class ShowHTML {
    public static void main(String [] args) throws Exception {
```

```
JEditorPane editor = new JEditorPane("text/html",
    "<H1>Wow!</H1><P><FONT COLOR=blue>That was easy</FONT></P>");
editor.setEditable(false);
JScrollPane pane = new JScrollPane(editor);
JFrame f = new JFrame("HTML Demo");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.getContentPane().add(pane);
f.setSize(800,600);
f.show();
    }
}
```

虽然你可以使用任何技术装载 HTML，并将它放到 JEditorPane 里，该组件实际上也能为你做到这一点。JEditorPane 的一个构造函数接受 String（另一个版本接受 URL 对象）表单的一个 URL。在清单 12.2 中可以看到这个例子。

清单 12.2 JEditorPane 可以直接载入一个 URL

```
import javax.swing.*;

public class ShowWeb {
    public static void main(String [] args) throws Exception {
        String url="http://www.yahoo.com";
        if (args.length>0) url=args[0];
        JEditorPane editor = new JEditorPane(url);
        editor.setEditable(false);
        JScrollPane pane = new JScrollPane(editor);
        JFrame f = new JFrame("HTML Demo");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.getContentPane().add(pane);
        f.setSize(800,600);
        f.show();
    }
}
```

如果你见过清单 12.2 中的小浏览器，就能看出超链接不做任何事情。另外，特殊的特征（如脚本以及动画工作的方式）同它们在完全的浏览器里不一样。

添加对超链接的支持比较容易。每次用户的指针移到超链接的上方时，离开超链接的区域，或者当用户点击超链接时，JEditorPane 可以触发一个事件。

为处理该事件，你必须在你的一类里实现 HyperlinkListener 接口，将该类的实例的一个引用传递给 JScrollPane.addHyperlinkListener 方法。清单 12.3 显示了一个可能的实现，我只为该类添加了 hyperlinkUpdate 方法，当该方法发现超链接激活时，它调用 setPage 方法移到新的一页，因为对 addHyperLinkListener 的调用需要一个对象实例，所以我将程序的逻辑从 main 方法移到 go 方法，接着在 main 里调用 go。否则，该程序与前一个程序很类似。

清单 12.3 可以使用 JEditorPane 处理超链接

```
import javax.swing.*;
import javax.swing.event.*;

public class ShowWeb1 implements HyperlinkListener {
    JEditorPane editor;
    public void hyperlinkUpdate(HyperlinkEvent ev) {
        if (ev.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
            try {
                editor.setPage(ev.getURL());
            }
            catch (Exception e) {} // what to do?
        }
    }

    public void go(String [] args) throws Exception {
        String url="http://www.yahoo.com";
        if (args.length>0) url=args[0];
        editor = new JEditorPane();
        editor.addHyperlinkListener(this);
        editor.setEditable(false);
        editor.setPage(url);
        JScrollPane pane = new JScrollPane(editor);
        JFrame f = new JFrame("HTML Demo");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.getContentPane().add(pane);
        f.setSize(800,600);
        f.show();
    }

    public static void main(String [] args) throws Exception {
        new ShowWeb1().go(args);
    }
}
```

JEditorPane 类也包含一个 read 方法, 允许你从 InputStream 里读 HTML。当输入来自网络而不是直接从一个 URL 中得来时, 这种方法就比较有用。

12.1.2 处理 HTML

有时你实际上可能要读 HTML 文件, 并且处理各个标签, 而不是简单地显示文本。可以使用下面几种方法, 当然, 你可以写一个很好的解析器。事实上, JavaCC 工具 (创建解析器的免费工具) 有几个 HTML 语法示例可用。但是, 使用这些工具需要一些关于解析器工作原

理的知识，同时需要了解如何去写正式的语法。

例如，假设你想从任意的网页上去掉格式，并使用有限的显示功能让它们在 PDA（掌上电脑）或者其他网络设备上更有用，你实际上不必太关心确切的文档结构，你只需要挑出一些关键标签，忽略注释，接着提取文本就可以了。

尽管你可以写一个正规的语法分析器，但看起来还是有点小题大做。一个更好的解决方法是写一个简单的 ad hoc（特别）解析器，可以读 HTML 文件，读的方式可能与人工读 HTML 的方式是一样的。这种技巧比较容易理解一些，不利的地方在于难以得到精确的结果，要做复杂的变化将有些困难。

12.1.3 实现 Ad Hoc

我决定创建一个解析器来接受 `InputStream`。这样，你可以解析一个文件、一个网站甚至任何你想转换成 `InputStream` 的东西。我只是需要一种更容易的方法来实现它，并不是指我不想重用已经写好的代码，因此我写了一个包含所有解析功能的通用类（见清单 12.4）。

清单 12.4 该类充当一个用作解析特别样式的网页的基类

```
import java.io.*;
import java.net.*;
import java.util.*;

public class AHParse {
    int lastchar=-2;
    StringBuffer current;

    public String parse(InputStream is) throws IOException {
        int c;
        int ender='<';
        int endoffset=0;
        boolean intag=false;
        boolean multicomment=false;
        int dashct=0;
        c=lastchar;
        current=new StringBuffer();
        if (c== -2) c=is.read();
        if (c<0) return null;
        if (c=='<') {
            intag=true;
            endoffset=1;
            ender='>';
            current.append((char)c);
            c=is.read();
            if (c=='!') {
                current.append((char)c);
                c=is.read();
            }
        }
    }
}
```

```

        if (c=='-') {
            current.append((char)c);
            c=is.read();
            if (c=='-') multicomment=true;
        }
    }
}
// read to end
while ((c!=ender && !multicomment)||
        (multicomment && c==ender && dashct!=2) ||
        (multicomment && c!=ender)) {
    current.append((char)c);
    c=is.read();
    if (c==-1) {
        endoffset+0;
        lastchar=-1;
        break;
    }
    if (lastchar=='-') dashct++; else dashct=0;
    lastchar=c;
}
while (endoffset--!=0) {
    current.append((char)c);
    lastchar=c=is.read();
}
return current.toString();
}

```

```

public void processURL(String urlstring) throws
    MalformedURLException, IOException {
    URL url=new URL(urlstring);
    InputStream is=url.openStream();
    String token;
    do {
        token=parse(is);
        // pass null to doElement to indicate EOF
        if (!doElement(token)) break;
    } while (token!=null);
    is.close();
}

```

```

// Override in subclass
public boolean doElement(String token) {
    if (token==null) return true;
    System.out.println(token);
    System.out.println("###");
    return true; // keep going
}

```

```

    }

    public static void main(String args[]) throws Exception {
        AHParse parser = new AHParse();
        parser.processURL(args[0]);
    }
}

```

当你想解析一个网页的时候，你只需要扩展这个通用类。可以将 URL 传给基类，它将为网页中的每个标签或者文本项重复调用 `doElement`。当然，那意味着你要重载 `doElement` 方法来执行任何你需要的处理。对 `doElement` 的每个调用都接收一个标签或者非标签项（可以区分两者之间的不同，因为标签是以圆括弧开始的）。最后，`doElement` 接收一个 `null`，以防你写任何关闭信息。

那么，解析 HTML 的特别规则是什么呢？首先，程序检查第一个输入字符，看它是否是圆括弧，如果是，则下一个输入标志是 HTML 标签；否则，它必须是某种文本。终止变量指定什么样的字符会终止该标志。在有标签的情况下，终止符是一个封闭的括弧；否则，终止符是一个开括弧（表示一个标签的开始）。

写这类解析器的一个问题是通常检查一个字符的次数太多。换句话说，终止符应该是下一个标志的开始。Java 中的 `PushbackReader` 类刚好用于这个目的，但是我决定在名为 `lastchar` 的变量里记录最后一个字符。

普通的文本很容易读，但是那些标签要识别注释就需要更多的工作。HTML 支持下面这种旧式的注释：

```
<!-- I am an old comment -->
```

它也支持 `<!--and end with -->` 这种新式的注释，如下所示：

```
<!-- I am a newer comment -->
```

旧式的注释像其他的标签一样。但是，新式的注释需要特殊的处理。解析器要读到一定程度才知道它处理的是什么样的注释，并且要相应地设置 `multicomment`。

12.1.4 Ad Hoc 细节

在预先的读操作完成之后，解析器不断循环，直到终止条件为止。因为使用新式注释 `while` 循环比你想象的要复杂一些：

```

while ((c!=ender && !multicomment) ||
       (multicomment && c==ender && dashct!=2) ||
       (multicomment && c!=ender)) {

```

从程序的英文明文中，可以看出，当 `multicomment` 为 `false` 时，终止符还没有读，要么当 `multicomment` 值为 `true` 时，终止符没有出现，要么它出现了，但是最后两个字符不是破折号。

很明显，程序必须对连续出现的破折号进行计数，也要跟踪变量 `endoffset`，它指出最后

读出的字符是标志的一部分还是下一个标志的开始。另外，你将注意到 `lastchar` 的值以 -2 开头，表明没有终止字符。我宁愿使用 -1，因为那表明文件的结束，我需要一个不同的值。

为提高程序效率，使用 `StringBuffer` 来创建标志。这比使用 `String` 效率要高，因为程序可以直接修改对象，而不用多个 `String` 对象。

实际上要从你的主程序里调用的方法只有 `parse` 方法，你可以重复调用它，直到它返回 `null` 值。然而，我想将整个逻辑流程封装在基类中，避免为不同的程序重复编码。

`processURL` 方法需要一个 `String` 类型的参数，这个参数包含了你要解析的 URL。它使用 URL 对象的 `openStream` 方法来获得一个 `InputStream`，与 URL 的文档相对应。缺省情况下，该方法只打印标志，但是你可以从新类继承，做任何你想做的事情。该类也有一个 `main` 例程，这样你可以在命令行里对它进行测试。

```
java AHParse http://www.coriolis.com
```

12.1.5 使用 AHParse

我的初始目标是将复杂的网页化简为一种更适合于仪表类型的形式，在 HTML 解析器的帮助下，这项工作相对简单。我的计划是将页面处理成下面的样子：

1. 忽略标准网页的标题。
2. 去掉 JavaScript 的链接。
3. 将标签 `<TABLE>` 以及 `<TR>` 转换成 `
` 标签。
4. 将 `<TD>` 标签转换成空格。
5. 将 `` 标签翻译成特殊的超链接。
6. 忽略 `<SCRIPT>` 标签中的任何内容。
7. 只传递 ``, `<PRE>`, `<P>`, `
` 和 `<A>` 标签（以及与它们相对应的结束标签）。
8. 允许图像列表中断，例如，用于格式化的 1 像素 GIF 文件。

完整的代码列在清单 12.5 中，程序相对比较直观，只有若干个难点。一个问题是 HTML 对大小写不敏感，但是，HTML 中的某些项如 URL，是大小写敏感的。`doElement` 方法将标志的大写复制，用于匹配文本。但是，当提取文本的一部分时，它使用初始的串值。

如果程序传递特殊的标签，它也应该传递相应的结束标签，这就是为何在检查传递标签之前，程序执行下述代码的原因：

```
if (tag.charAt(0)!='/') tag=tag.substring(1);
```

这样，余下的程序可以只检查基本的标签，换句话说，在该行执行之后，标签 `/A` 和 `A` 都与 `A` 匹配。

清单 12.5 程序 WebParse 将网页转换成简单的页面

```
import java.util.*;

public class WebParse extends AHParse {
    boolean first=true;
```

```

boolean inscript=false;
boolean inanchor=false;
static String page;
static Properties imageIgnore = new Properties();

public void doHead() {
    System.out.println("<HTML><HEAD><TITLE>Clip from " + page+"</TITLE>");
    System.out.println("<BASE HREF=\"\" + page + \"\">");
    System.out.println("</HEAD><BODY>");
}

public void doEnd() {
    System.out.println("</BODY></HTML>");
}

// Assumes string is upper
private String extractAttribute(String token,String tag,String defval) {
    String utoken=token.toUpperCase();
    int n=utoken.indexOf(tag),n1,n2;
    if (n== -1) {
        return defval;
    }
    char match = ' ';
    n+=tag.length();
    if (utoken.charAt(n)=='"') {
        match='"';
        n++;
    } else if (utoken.charAt(n)=='\'') {
        match='\'';
        n++;
    }
    if (match==' ') {
        n1=utoken.indexOf(' ',n);
        n2=utoken.indexOf('>',n);
        if (n1== -1 || (n2!= -1 && n2<n1)) n1=n2;
    }
    else
        n1=utoken.indexOf(match,n);
    if (n1== -1) return token.substring(n); // technically an error!
    return token.substring(n,n1);
}

public boolean doElement(String token) {
    if (token==null) {
        doEnd();
    }
}

```



```

        return true;
    }
    if (first) doHead();
    first=false;
    if (token.charAt(0)=='<') {
        // tag
        boolean pass=false;
        String utoken=token.toUpperCase();
        String tag = new StringTokenizer(utoken.substring(1),
            " \t>").nextToken();
        if (tag.equals("A")) {
            String hrefurl=extractAttribute(token, "HREF=", "");
            if (!hrefurl.equals("")) {
                if (hrefurl.length()>10 &&
                    hrefurl.substring(0,10).compareToIgnoreCase("JAVASCRIPT")!=0)
                    return true; // ignore javascript links
            }
            inanchor=true;
        }
        if (tag.equals("/A")) inanchor=false;
        if (tag.equals("TABLE")) System.out.println("<BR>");
        if (tag.equals("/TD")) System.out.println("&nbsp;&nbsp;&nbsp;");
        if (tag.equals("/TR")) System.out.println("<BR>");
        if (tag.equals("IMG")) {
            int n;
            String src=extractAttribute(token, "SRC=", "");
            if (src.equals("")) return true; // ???
            n=src.lastIndexOf('/');
            String srcbase = n== -1?src:src.substring(n+1);
            if (imageIgnore.get(srcbase)!=null) return true;
            String alt=
                extractAttribute(token,
                    "ALT=", srcbase);
            if (alt.equals("")) alt=srcbase; // real ALT=""
            if (!inanchor) {
                System.out.println("<A HREF=\"\" +
                    src + \"\">&lt;&lt;&lt;Image: "
                    + alt + "&gt;&gt;&gt;</A>");
            }
            else {
                System.out.println("&lt;&lt;&lt;Image: " + alt +
                    "&gt;&gt;&gt;");
            }
        }
        return true;
    }
    if (tag.equals("SCRIPT")) inscript=true;
    if (tag.equals("/SCRIPT")) inscript=false;
    if (tag.charAt(0)=='/') tag=tag.substring(1);

```

```

        if (tag.equals("B")||tag.equals("I")||tag.equals("U")) pass=true;
        if (tag.equals("PRE")) pass=true;
        if (tag.equals("P")) pass=true;
        if (tag.equals("BR")) pass=true;
        if (tag.equals("A")) pass=true;
        if (pass) System.out.println(token);
    }
    else {
        // text
        if (!inscript) System.out.println(token);
    }
    return true;
}

public static void main(String [] args) throws Exception {
    page=args[0];
    for (int n=1;n<args.length;n++)
        imageIgnore.put(args[n],"y");
    new WebParse().processURL(page);
}
}

```

12.1.6 处理图像

我不想页面中的图像比较混乱，但是我还是想让用户在他们想看的时候能够看到图像。解决方法就是将标签转换成超链接。程序中使用超链接的文本与图像的 ALT 属性等价，或者如果没有文本，就使用图像文件名来代替。

我注意得比较早的一件事是有很多网站使用图作衬托，而且页面上到处都有，但是在只有文本的情况下毫无意义。所以，我添加了一个列表来查找图像的基本名字，如果图像名在列表里，程序就将图像从最终输出里删除，程序从任意附加的命令行参数里读这个列表。

链接到一个图像的另一个问题产生在图像的超链接是其本身的时候，我修改程序来检测它什么时候在一个超级链接里（inanchor 标志）。当该标志为真时，程序将不发出该图像的链接，只链接对应于该图像的文本。这防止了嵌套链接（比较容易混淆）。

12.1.7 属性解析

尽管 AHParse 做了绝大部分工作，有一项解析工作仍然是它的基类做的。为了正确地传输超链接和图像，程序需要提取属性值，如 HREF 和 SRC。这就是 extractAttribute 方法的。该方法需要三个参数：一个串、一个要提取的大写的属性名以及一个在没有属性时使用的缺省值。这样你可以写如下代码：

```
String hrefurl=extractAttribute(token,"HREF=","NoLink.htm");
```

注意 HTML 的作者可以提供空属性（如 HREF=""），这样将不返回缺省串，因为该属性没有忽略，它只是为空。

解析属性值带有一定技巧，因为有三种情况要处理：

- 没有空格的属性值不需要引号（HREF=x.htm）。
- 值可能被单引号包含（HREF= 'x.htm'）。
- 值可能被双引号包含（HREF= "x.htm"）。

代码假设未加引号的属性将以一个空格或者闭括弧终止，任何加引号的属性应该以相匹配的引号来终止。

12.1.8 改进可用性

在使用 WebParse 类进行一段时间的试验之后，我开始考虑如何让它更有用。很明显，考虑到图像的超链接的程序逻辑不容易改变。但是，如果有更简单的方法来设置程序传递什么标签，那将是一个很好的改进。

结果是采用 WebParse2（见清单 12.6）。该版本需要两个命令行参数——网站以及含有待传递的标签的属性文件名。附加的命令行参数像以前一样构成图像的排除列表。

清单 12.6 这个版本的 Web 解析器更加灵活，因为它允许动态选择传递给输出的标签值

```
import java.util.*;
import java.io.*;

public class WebParse2 extends AHParse {
    boolean first=true;
    boolean inscript=false;
    boolean inanchor=false;
    static String page;
    static Properties imageIgnore = new Properties();
    static Properties passtag = new Properties();

    public void doHead() {
        System.out.println("<HTML><HEAD><TITLE>Clip from " + page+"</TITLE>");
        System.out.println("<BASE HREF=\"\" + page + "\">");
        System.out.println("</HEAD><BODY>");
    }

    public void doEnd() {
        System.out.println("</BODY></HTML>");
    }

    // Assumes string is upper
    private String extractAttribute(String token,String tag,String defval) {
        String utoken=token.toUpperCase();
        int n=utoken.indexOf(tag),n1,n2;
        if (n!=-1) {
```

```

        return defval;
    }
    char match = ' ';
    n+=tag.length();
    if (utoken.charAt(n)=='"') {
        match='"';
        n++;
    } else if (utoken.charAt(n)=='\'') {
        match='\'';
        n++;
    }
    if (match==' ') {
        n1=utoken.indexOf(' ',n);
        n2=utoken.indexOf('>',n);
        if (n1!=-1 || (n2!=-1 && n2<n1)) n1=n2;
    }
    else
        n1=utoken.indexOf(match,n);
    if (n1==-1) return token.substring(n); // technically an error!
    return token.substring(n,n1);
}

public boolean doElement(String token) {
    if (token==null) {
        doEnd();
        return true;
    }
    if (first) doHead();
    first=false;
    if (token.charAt(0)=='<') {
        // tag
        boolean pass=false;
        String utoken=token.toUpperCase();
        String tag = new StringTokenizer(utoken.substring(1),
            " \t>").nextToken();

        if (tag.equals("A")) {
            String hrefurl=extractAttribute(token,"HREF=", "");
            if (!hrefurl.equals("")) {
                if (hrefurl.length()>10 &&
                    hrefurl.substring(0,10).compareToIgnoreCase("JAVASCRIPT")==0)
                    return true; // ignore javascript links
            }
            inanchor=true;
        }
        if (tag.equals("/A")) inanchor=false;
        if (tag.equals("TABLE")) System.out.println("<BR>");
        if (tag.equals("/TD")) System.out.println("&nbsp;&nbsp; ");
        if (tag.equals("/TR")) System.out.println("<BR>");
    }
}

```

```

        if (tag.equals("IMG")) {
            int n;
            String src=extractAttribute(token, "SRC=", "");
            if (src.equals("")) return true; // ???
            n=src.lastIndexOf('/');
            String srcbase = n==-1?src:src.substring(n+1);
            if (imageIgnore.get(srcbase)!=null) return true;
            String alt=
                extractAttribute(token,
                    "ALT=", srcbase);
            if (alt.equals("")) alt=srcbase; // real ALT=""
            if (!inanchor) {
                System.out.println("<A HREF=\" " +
                    src + "\">&lt;&lt;&lt;Image: "
                    + alt + "&gt;&gt;&gt;</A>");
            }
            else {
                System.out.println("&lt;&lt;&lt;Image: " + alt +
                    "&gt;&gt;&gt;");
            }
            return true;
        }
        if (tag.equals("SCRIPT")) inscript=true;
        if (tag.equals("/SCRIPT")) inscript=false;
        if (tag.charAt(0)=='/') tag=tag.substring(1);
        if (passtag.get(tag)!=null) pass=true;
        if (pass) System.out.println(token);
    }
    else {
        // text
        if (!inscript) System.out.println(token);
    }
    return true;
}

public static void main(String [] args) throws Exception {
    page=args[0];
    for (int n=2;n<args.length;n++)
        imageIgnore.put(args[n], "Y");
    FileInputStream file=new FileInputStream(args[1]);
    passtag.load(file);
    new WebParse2().processURL(page);
}
}

```

12.1.9 再次访问 Swing

在本章的前边，你看到如何使用 Swing 组件来显示 HTML，它坚持 Swing 中的有些方法

可以用于解析 HTML，如果你能想出如何使用 Swing 解析器，你就不必自己亲自写代码了。

类 JEditorPane 使用类 javax.swing.html.HTMLEditorKit.Parser 来解析 HTML。该类实际上是抽象类，但是类库里提供了在 javax.swing.text.html.parser.ParserDelegator 类中的一个实现。

该类从 Reader 流里接收输入并查找任何下述内容：

- 起始标签
- 结束标签
- 简单标签
- 文本
- 注释

这些都是 HTML 文件中最有可能对你的程序有用的部分。当解析器找到其中一项时，它调用你提供的回调方法，这样你可以作任何操作。

解析器不认可的标签仍然产生调用（像对任何简单标签一样）。另外，你可以向解析器请求标签拥有的任何属性（例如，<A>标签的一个 HREF 属性）。

清单 12.7 显示了一个简单的程序，使用 Swing 解析器来产生一个网页，它包含到一个特定网页的链接。

清单 12.7 该程序使用 Swing 来解析 HTML

```
import javax.swing.text.*;
import javax.swing.text.html.*;
import javax.swing.text.html.parser.*;
import java.net.*;
import java.io.*;

public class LinkPage extends HTMLEditorKit.ParserCallback {

    public void handleStartTag(HTML.Tag t,
        MutableAttributeSet a,
        int pos) {
        if (t== HTML.Tag.A) {
            System.out.println(
                "<A HREF=\"" +
                a.getAttribute(HTML.Attribute.HREF)+
                "\">" +
                a.getAttribute(HTML.Attribute.HREF) +
                "</A><BR>");
        }
    }

    public static void main(String args[]) throws Exception {
```

```
URL url=new URL(args[0]);
Reader reader = new InputStreamReader((InputStream)url.getContent());
System.out.println("<HTML><HEAD><TITLE>Links for " + args[0] +
    "</TITLE>");
System.out.println("<BASE HREF=\"' + args[0] + '\"></HEAD>");
System.out.println("<BODY>");
new ParserDelegator().parse(reader, new LinkPage(), false);
System.out.println("</BODY></HTML>");
}
}
```

该对象扩展了 `HTMLEditorKit.ParserCallback`。因为它只需要处理标签 `<A>`，子类只要重载 `handleStartTag`。并且，程序只关心 `<A>` 标签。这样它要将参数 `HTML.Tag` 同 `HTML.Tag.A` 进行比较。如果有一个匹配，它就检查提供的 `MutableAttributeSet` 对象，得到 `HREF` 属性的值。

`main` 例程为目标 URL 简单地打开一个流，将其转换成一个 `Reader`，并将它传给 `ParserDelegator.parse` 方法，除了输出的开头和结尾中的某些样板文本以外，程序从回调函数里创建输出。

12.2 快速解决方案

12.2.1 通过 Swing 使用 HTML

用于程序中的 Swing 库常用于需要高级用户接口的程序中。大多数 Swing 组件，如标签，将接受一些 HTML 来改变它们的文本显示。例如：

```
JLabel lbl = new JLabel("<HTML><I>Italics are easy</I></HTML>");
```

但是，像在下一节里你将看到的一样，可以使用 Swing 的 `JEditorPane` 组件来完整地显示 HTML 页面。

12.2.2 使用 JEditorPane 显示 HTML

`JEditorPane` 可以根据 URL 显示复杂的 HTML 页面，你可以采用几种方式对该组件进行初始化：

- 使用带两个参数的构造函数来创建 `JEditorPane` 对象，第一个参数是 MIME 的类型 (`text/html`)，第二个参数是待显示的 HTML。
- 使用一个 URL 对象或者是代表 URL 的 String 来创建 `JEditorPane` 对象。
- 调用 `setPage` 来载入一个特定的 URL。
- 先调用 `setContentType` (传递 `text/html`)，接着调用 `setText`。
- 使用 `read` 方法将 `JEditorPane` 传递给一个 `InputStream`。

你可以在清单 12.1 和 12.2 中找到相关示例代码。`JEditorPane` 组件的构造函数比较直观：

```
JEditorPane editor = new JEditorPane(url);
editor.setEditable(false);
```

对 `setEditable` 的调用阻止用户改变编辑器的内容。

12.2.3 通过超链接显示 HTML

如果你使用 `JEditorPane` 来显示 HTML，可能要对超链接进行相关处理，在标准组件中是没有相关处理的。你可以提供一个实现了 `HyperlinkListener` 接口的对象，它在用户指针移到一个超链接区域中、移出区域或者激活超链接的时候，执行某些操作。

这个接口只有一个方法，即 `hyperlinkUpdate`。该方法接收一个参数，它是一个 `HyperlinkEvent` 对象。使用该对象，你可以知道超链接的 URL 以及编辑器上所发生的事件。下面是这个方法的一个例子：

```
public void hyperlinkUpdate(HyperlinkEvent ev) {
    if (ev.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
        // perform the action you want when the user clicks the hyperlink
    }
}
```

清单 12.3 中的完整实现使用了 `setPage` 方法来将显示移动到新的 URL。

12.2.4 使用 AHParse

如果你想存储一个 HTML 文件的内容含义，你可以使用自己读这个 HTML 的方式来拆分该 HTML 文件。这种类型的解析方法称为 ad hoc 解析，容易理解和实现。清单 12.4 显示出一个可能的实现。

在下述方法中，类 `AHParse` 将 HTML 文件拆分，并能够正确地说明其中的引用串以及注释。

- 解析器遇到的每个 HTML 元素都会引起对 `doElement` 方法的调用。
- 该方法的代码可以区分标签以及普通文本，因为标签是以尖括弧开头的。
- 当文件结束时，解析器使用参数 `null` 来调用 `doElement` 方法。

要编一个解析 HTML 文件的程序，将只能对 `AHParse` 类进行扩展：

```
public class myParser extends AHParse { . . .
```

在子类里，你要实现执行所有你需要的任务的方法。清单 12.5 和清单 12.6 中的例子，将复杂的页面解析成更简单的而且等价的页面。

12.2.5 通过 Swing 解析标签

Swing 库提供了 Swing 组件内部使用的 HTML 解析器。但是，如果你知道如何让它工作，你也可以使用这个解析器。Swing 用作解析器的类是 `javax.swing.text.html.parser.ParserDelegator`。它从 `Reader` 流里接受输入，并查找 HTML 文档的各个不同部分。当它找到一个元素时，它调用你在类里提供的一个相应的方法：

- 起始标签——`handleStartTag`

- 终止标签——handleEndTag
- 简单标签——handleSimpleTag
- 文本——handleText
- 注释——handleComment
- 错误信息——handleError

清单 12.8 的例子中显示了对一个网页上不同元素的报告。你将注意到 Swing 解析器将未知的标签看作是简单的标签，即使它们确实有一个结束标签。另一个奇怪现象是解析器会添加标签，形成正确的文档。例如，如果文档没有<HTML> 或 <BODY>标签，解析器将产生这些标签，即使它们没有在文档里出现也将它们报告出来。

清单 12.8 该程序打印关于网页中各个元素的报告

```
import javax.swing.text.*;
import javax.swing.text.html.*;
import javax.swing.text.html.parser.*;
import java.net.*;
import java.io.*;

public class HTMLParse extends HTMLToolkit.ParserCallback {
    public void handleText(char[] data, int pos) {
        System.out.println(data);
    }

    public void handleStartTag(HTML.Tag t,
        MutableAttributeSet a,
        int pos) {
        System.out.println("+" + t.toString());
    }

    public void handleSimpleTag(HTML.Tag t,
        MutableAttributeSet a,
        int pos) {
        System.out.println("*" + t.toString());
    }

    public void handleEndTag(HTML.Tag t, int pos) {
        System.out.println("-" + t.toString());
    }

    public static void main(String args[]) throws Exception {
        URL url=new URL(args[0]);
        Reader reader = new InputStreamReader((InputStream)url.getContent());
        new ParserDelegator().parse(reader, new HTMLParse(), false);
    }
}
```

12.2.6 通过 Swing 解析属性

当然，有些标签拥有属性。例如，标签就有一个 SRC 属性，表示图像文件名。HandleStartTag 和 handleSimpleTag 方法都接收 MutableAttributeSet 对象（你可以用来查找属性值）作为参数。例如：

```
public void handleStartTag(HTML.Tag t,
    MutableAttributeSet a,
    int pos) {
    if (t== HTML.Tag.IMG) {
        System.out.println(
            "Source = " +
            a.getAttribute(HTML.Attribute.SRC));
    }
}
```

除了常规的 HTML 属性外，解析器添加了附加的属性来分解有关解析处理（你可能需要了解）的信息。例如，当解析器产生伪标签来替代遗漏的标签时，它就设置 ParserCallback.IMPLIED 属性值。一个未知的结束标签将拥有 HTML.Attribute.ENDTAG 属性。

12.2.7 通过 Swing 解析文本

尽管你常常想用 Swing 解析器来对标签进行解析，Swing 解析器也要接收 HTML 文档中的文本，你只需要提供对 ParserCallback 类中的 handleText 方法的一个实现。清单 12.9 显示了一个简单的例子：

清单 12.9 从 HTML 文档中提取文本

```
import javax.swing.text.*;
import javax.swing.text.html.*;
import javax.swing.text.html.parser.*;
import java.net.*;
import java.io.*;

public class TextOnly extends HTMLToolkit.ParserCallback {
    public void handleText(char[] data, int pos) {
        System.out.println(data);
    }

    public static void main(String args[]) throws Exception {
        URL url=new URL(args[0]);
        Reader reader = new InputStreamReader((InputStream)url.getContent());
        new ParserDelegator().parse(reader, new TextOnly(), false);
    }
}
```


第 13 章 HTML 服务

13.1 深入介绍

Reese 的花生酱黄油奖杯——是 Hershey 公司的长期产品，该公司发生过一次著名的广告战，是由一个巧克力爱好者与一个花生黄油爱好者的纠纷引起的。我怀疑那次冲突是否真的发生了，但它有可能在网上已经发生过。Web 是两种老技术的结合：互联网络和超文本。

如果谈到你的那些不懂计算机技术的邻居或者亲戚，你可能发现他们认为 Web 就是因特网（Internet）。当然，Internet 比 Web 要早出现几年。Vannervar Bush 在 1945 年首先预想了超文本的概念，这已不再是新思想了。的确，与 1976 年相比，现在有更多的人上 Internet，而且有更多的在线信息可以获取。但是，Web 并不能让人们共享新信息，而只是让他们非常方便地共享信息。

毫无疑问 Web 使得 Internet 可以访问，并且为它的爆炸性增长起了催化作用。但有趣的是，最复杂的地方在于客户端的均衡。客户端（也就是 Web 浏览器）不得不处理显示格式、超链接、样式单和无数的其他的问题。最简单的服务器只是接受请求，发送带有一些基本标题的文件。

当然，流行的服务器可能不得不做更多的工作。现在的服务器可以包括脚本语言、安全特征以及复杂事务管理系统的接口。并且，对大容量的 Web 服务器而言，性能是很关键的。

你要写一个具备各种功能的大型 Web 服务器程序是不太可能的。这些程序可以从那些大供应商那里获得，有几个流行的 Web 服务器程序还是免费的。但是，知道如何写小型的 Web 服务器程序是很有用的。专用服务器能够满足很多功能，甚至可以帮助你为大型系统调试错误。

如果想写一个 Web 服务器程序，第一个问题是：“有必要吗？”很多 Web 服务器现在都支持 JSP 和 servlet。使用 JSP 和 servlet，可以写成在现存的 Web 服务器上运行的 Java 小程序。这样，你不用照抄服务器的核心逻辑流程就可以做你想做的事情。

但是，有时候为了某种特殊的任务，你也需要写一个服务器程序或者使用开放源码服务器。幸运的是，简单的服务器不会那么复杂，这在本章里就可以看到。

13.1.1 关于 JSP

Java 一个最强大的地方也是它首要的弱点：即处处可运行。的确，Java 的跨平台特性让它很有吸引力，但是它也要求每个用户都正确地安装了 Java 虚拟机，按照你的想法进行工作。有时这可能是一个关键的次序。用户拥有 Swing 或者最新版的 Java 虚拟机吗？我最近通过对

Java 小程序签名的处理发现所有的 Web 浏览器都需要一个不同的处理过程，并且有些浏览器根本不允许签名。

你可以通过运行所有你自己的 Java 程序来控制平台问题，也就是说，通过使用 Java 来产生 HTML 页面，并将它们发送到客户端。这样的服务器端程序，不管是用 Java 还是其他技术，已经成为很多网站的一种技术选择。你可以控制环境，确保程序得到正确的运行。

将 Java 融合到 Web 服务器端的一个方法是使用 servlet，但是，servlet 是专门的 Java 程序（与客户端的 applet 小程序类似），并且写 servlet 程序有些复杂。

这就是为何这里要提 JSP，JSP 可以让你在 HTML 脚本中嵌入 Java 代码，服务器或者服务器的特殊扩展将 JSP 文件编译成需要的 servlet。这是使用 servlet 的一种简单方式，几乎所有人都可以创建 JSP 程序。JSP 系统将 JSP 脚本程序都编译成 servlet 程序。

假设你想创建简单的模板系统，使得能将文本文档格式化成网页，这样它们拥有一致的外观。这个系统将会让满意的开发人员将信息集中在页面上，同时集中的思想在于确定页面的总体样式和设计。

如果你为一个俱乐部、学校或者甚至是一个公司搭起一个网站，访问网站的人都想创建自己的内容。但是你还强制实行站点的专门样式，这才是比较理想的。通过在系统中融入 JSP，让用户上传他们自己的文件，可以很容易地创建一个俱乐部成员花名册或者其他类似的站点。

通过一些修改，你也可以把站点放到网上，通过文件或者数据库来驱动。当然，你另外还需要一些数据。文件的头几行可以指定各项的图片文件、价格以及其他的与目录相关的数据。你甚至可以解析 XML 语言来产生目录页。

13.1.1.1 你需要的是什么

如果你想启动服务器端的 Java，那就需要一个 Web 主机支持 JSP。幸运的是，几乎所有服务器都可以采取某些方法来与 JSP 结合。但是，如果在你的产品服务器上试运行，你可能感到有些疑虑。如果碰到这种情况，可以免费获得微软的个人 Web 服务器（PWS），添加上 Allaire 的 JRun 的 demo 版。这是测试 JSP 程序的好开端，但是你不必将真实的网站运行在这些工具上，因为它们的并发连接数有限制。但是，这对于开发来说是再好不过了，因为你不必付任何价钱。

也可以从其他服务器那里得到若干免费的 JSP 插件。你可以在 www.serverpages.com 那里找到完整的列表。甚至著名的 GNU 工程也有一个 JSP 包，使用了很多流行的 servlet 插件（例如 Apache 中的 Jserve 插件）。Apache 工程中的 Tomcat 也非常流行。

13.1.1.2 就绪、设置...

一旦你有了能够处理 JSP 的服务器，就可以试着写些简单的 JSP 脚本。记住 JSP 页要经过严格的 Java 编译器。（同时不要混淆：这里不是 JavaScript。）创建一个名为 test0.jsp 的文件，包含下述所有代码：

```
<%
```

```
java.util.Date dt = new java.util.Date();
out.println(dt.toString());
%>
```

如果你怀疑这看起来有些像微软的 ASP，那就对了，JSP 和 ASP 有着惊人的相似之处（从 <http://java.sun.com/products/jsp/jsp-asp.html> 那里可以了解到）。但是，尽管 ASP 解释 JavaScript、VBScript 或者 PerlScript，JSP 编译的是真正的 Java 代码。

你可以猜猜 test0.jsp 文件做些什么。使用缺省的构造函数创建一个新的 Date 对象，让你可以访问当前时间和日期。使用 toString 将时间和日期转换成人们可读的串值。而 out 对象是几个内建对象之一，可以在写 JSP 脚本时使用。就像名字本身所指一样，out 对象让你在脚本中输出 HTML 代码。

注意 JSP 文件同时包含有 Java 和 HTML。当浏览器看到文件时，没有别的，只有 HTML。你甚至可以将 Java 与 HTML 混合在一起，像下面这样：

```
<%
    java.util.Date dt = new java.util.Date();
    out.println(dt.toString());
    int n=dt.getDay();
%>
<BR>
<%

    if (n==0) {
%>
        The <b>weekend</b> is almost over!
<%
    }

    else if (n==6) {
%>
        The <b>weekend<b> is here!
<%
    }

    else if (n==5)
%>
        TGIF
<%
    }
    else
    {
%>
        Another weekday
<%
    }

%>
```

这个 JSP 脚本迎合了那些带有指定时间的消息的用户（当然，那是服务器上的时间，不是运行浏览器的机器上的时间）。

13.1.1.3 一个有用的工程

使用 JSP 为普通文件内容服务是相当简单的。系统的核心是 index.jsp（见清单 13.1）。该页本身是一个微型服务器程序。它检查 doc 查询串来确定显示哪个文档（或者缺省情况下使用 index.txt）。它打开文档并将其显示成预定的格式。

清单 13.1 该 JSP 程序使用文本文件和预定义的样式

```
<%@ page import="java.io.*" %>
<%@ page import="java.util.*" %>
<%
    FileReader r;
    String s;
    String errm = ""; // error message?
// find file name
    String doc= request.getParameter("doc");
// if no doc use index.txt
    if (doc==null) doc="index.txt";
    try {
        r = new FileReader(application.getRealPath(doc));
    }
    catch (IOException e) {
// whoops set error message and open error.txt instead.
        errm = e.getMessage();
        r = new FileReader(application.getRealPath("error.txt"));
    }
    BufferedReader br=new BufferedReader(r);
    s=br.readLine(); // get title
%>
<html>
<head>
<title><%= s %></title>
</head>
<body bgcolor=#FFFFFF>
<H1><%= s %></H1>
<%
    if (errm!="") {
        out.println(errm);
        out.println("<BR>");
    }
    boolean ul=false;
    do {
        s=br.readLine();
        if (s!=null && s.length()!=0) {
            if (s.charAt(0)=='*') {
```

```
        if (!ul) out.println("<UL>");
        ul=true;
        out.println("<LI>");
        s=s.substring(1);
    }
    else if (ul==true) {
        ul=false;
        out.println("</UL>");
    }
}
if (s!=null) out.println(s);
) while (s!=null);
r.close();
// produce footer
%>
<hr>
<%- new Date().toString() %><BR>

</body>
</html>
```

文档文件都是包含相关内容的纯文本文件。第一行由 JSP 脚本解析，用于产生网页的标题。文档文件的剩余部分是普通的 HTML，其中的例外是你不能使用重要的标签，如<HTML>、<HEAD> 或者 <BODY>。这些标签都帮助定义文档的结构，不是因为 JSP 脚本为我们做这些事，在文档文件里我们没有必要也不需要它们。除此之外，你可以使用<P>和
等标签用于基本的文档格式。该脚本本身并不希望在文档文件里出现行中断（这与浏览器对待 HTML 文件一样），但是如果要修改 index.jsp，让它来做这件事也很容易。

JSP 文件执行文件本身的唯一不寻常的地方是当它发现以星号开头的行时，它对任意这样的行都是产生恰当的 HTML。你可以对 index.jsp 文件作出任意多的变化，以改变网页的外观或者改变脚本处理文件的方式。

如果你习惯于处理小程序（applet），你可能惊奇地发现 JSP 脚本可以执行读文件操作。记住，JSP 脚本要转变成在服务上运行的严格的 Java 程序，而不是一个在浏览器上运行的 applet 小程序。那意味着适用于 applet 的所有常用安全限制对 JSP 脚本都不起作用（或者是 servlet）。

13.1.1.4 index.jsp 内幕

最开始，程序 index.jsp 试图打开文档文件。为做到这一点，它首先需要读 URL 请求末尾的查询串。JSP 提供了 request 对象，可以让你从表单变量以及其他你认为是输入的串里读查询串。语句 request.getParameter("doc") 返回文档名（可能为空）。如果返回为空，就意味着访问者访问到你站点中不存在的 URL，程序将变量 doc 值设为 index.txt，这是缺省的文档（见清单 13.2）。

清单 13.2 一个文本模板示例

An example page


```
This page is an example of the JSP presentation framework.  
You can <i>still</i> use HTML tags, and in fact you must use  
them to create line breaks and other special elements.  
<BR>  
Features:<BR>  
* Central administration of page look and feel  
* Addition of standardized headers, footers, or table elements  
* Automatic lists (like this one)  
<BR>  
Have fun! -- Al Williams  
<BR>  
<A HREF=?doc=test2.txt>Another automatic page</A>
```

如果 doc 变量里有数据, 程序将使用 `application.getRealPath` 方法构造文档的路径。这个方法采取的是文档的虚拟路径 (像 `/test.txt` 一样), 并返回文档的物理路径 (如 `c:\inetpub\wwwroot\test.txt`)。该脚本使用完全的路径名来构造一个 `FileReader` 对象。如果文件名或者访问文件有错误, 这个调用就会抛出异常。在通常的程序或者小程序 (applet) 里, 必须额外编写代码来处理异常 (或者至少要声明函数可能要通过自己抛出异常来传递错误)。但是, JSP 脚本运行在捕获所有异常的包里。因此如果你的程序抛出一个异常, JSP 系统会将它报告给用户。

当然, 如果你不想让用户看到原始的错误信息, 或者是为了某种原因, 在程序继续执行之前要对异常进行相关处理时, 你还可以自己写代码来捕获异常。在我的脚本里, 我已经决定处理异常, 这样就可以返回 `error.txt` 文件——一个标准的错误处理页面——而不是用户试图指定的页。当然, 你可以在 `error.txt` 文件里放入任何东西, 来告知用户有错误发生。

一旦 `FileReader` 对象构造完成, 它指向一个文件: 或者是请求的文件, 即缺省的 `index.txt` 文件, 或者是 `error.txt` 文件。脚本使用该对象来构造一个 `BufferedReader` 对象, `BufferedReader` 的 `readLine` 方法每次从指定的文件中选择一行。

脚本读到的第一行用做形成 HTML 页面的标题。脚本在 `<TITLE>` 标签内写这一行, 在正文部分使用 `<H1>` 标签也写这一行。脚本中使用特殊的 JSP 标签 `<%= %>` 来实现这两个任务。在 JSP 中使用这个标签与使用 `out.println` 类似。在 `<%=` 与 `%>` 之间不管出现什么内容都要考虑, 并会出现在结果 HTML 中。注意, 在表达式末尾不必使用分号, 并且标签内只能有一个表达式。

在脚本输出标题之后, 当打开 `FileReader` 对象时, 它输出可能产生的任何错误。一旦前边都正确无误, 脚本将继续逐行执行下去。如果一行开头出现星号, 脚本会使用 `` 标签来开始一个新列表 (除非已经有一个列表, 那么它就继续使用当前列表)。脚本对开头带星号的每行也加上 `` 标签。如果脚本经过没有星号的行, 它假设该列表结束, 并输出一个结束的 `` 标签。

一旦脚本处理了所有的行, 它就输出由水平线 (`<HR>`) 组成的页脚和当前的日期时间。当然, 你也可以改变它, 这是脚本的整个亮点, 它可以让你很容易地定制适用于每一页的模

板的外观。

13.1.2 定制 Tandem 中的服务

在第 10 章，我们看到了如何写 Java 中的定制服务器，可以代表 Java applet 处理特殊的任务。那个例程（一个 tic-tac-toe 游戏）使用了定制服务器，连接不同机器上的 applet。

这对 applet 工作得很好，但有时 applet 不是解析问题的最佳答案。在现在的客户机/服务器编程模式下，很显然 Web 浏览器是最终的客户端。不仅是所有人都使用 Web 浏览器，包括那些大公司像微软，都投入大量的钱用于它们的开发。从我个人预算来看，我不太可能去做出一个客户端软件来与微软或者网景（Netscape）的客户端软件相抗衡。

看看一个非常简单的例子。假设你的销售部门想要从你的个人计算机上输入订单到大型机上的订单入口系统，有好几种方法可以选择，但为简化问题，进一步假设你要将输入存到服务器上一些以逗号定界的文件里头。假设大型机要批处理地读这些文件。

这个并不太难，你可以使用 Perl 来实现它，当然，任何 CGI 脚本语言都可以实现这个任务。ASP 能够做到，Java Servlet 也能做到。但是，所有这些方法都预先假定一件事：你能够与服务器相连。换句话说，只要销售人员在办公室（网络正常），就不会有问题。但是如果他们在外边并且连接不到服务器，那该怎么办呢？

在这种情况下，客户端软件应该在用户的硬盘上写一个本地文件。听起来很好，但如何实现呢？通常的网页，包括通常的 applet，是不能向用户的硬盘中写东西的。你可以给 applet 签名，但是它需要执行两个独立的函数：使用网络连接向服务器写，向硬盘做写操作。这也假设了用户允许签名的 applet 能够向硬盘写，但这并不一定有效。

ActiveX 能够做到上述这点，就大多数而言，只能在 IE 下工作。你如何设计一种简单的方法来在服务器或者本地硬盘上存储文件呢？

一个可能的答案是在客户端计算机上写一个定制的 Web 服务器。Java 实现了管理网络连接的绝大部分工作。并且，网上有大量的服务器应用的例子。你唯一要做的工作是让服务器做你需要的专门处理。当在网络中运行时，使用常规的服务器，但当不在网络环境下运行时，可以使用本地服务器。

为何不使用正规的本地 Web 服务器呢？首先，完全的服务器需要的软件通常比膝上电脑用户想安装的软件还要多。其次，还有维护以及许可权这些令人头痛的问题。如果运行自己的服务器足够容易，为何不运行自己的呢？

不要误解其中的意思，我们一般不太可能花时间写一个可以与 Apache 或者 WebSphere 相抗衡的服务器软件。但是，反过来想，你真的有必要那么做吗？几乎没有这个必要。

13.1.2.1 计划

为简化问题，我为服务器设计了若干基本原则。首先，它不必支持脚本、CGI 或者 POST 方法，只需要支持老式的 HTTP1.0 中的 GET 方法。其次，我想让服务器识别出 CGI 脚本的请求，并使用 Java 作相应的处理。显然，如果你可以很容易地向服务器添加常用函数，那就

再好不过了。

我写的伪 CGI 脚本只是查找一个名为 `Filename` 的表单入口（大小写是很重要的）。如果脚本发现这个入口，它假设要将所有其他入口都写成一种逗号隔开的格式，并将它追加到 `Filename` 指定的文件里。通常，`Filename` 是表单的隐藏域，通过运行在浏览器里的 JavaScript 设置成合适的值。

13.1.2.2 第一步：站在别人的肩膀上

有关 Internet 的一件大事是有人很可能已经写出了你能想像得到的东西。如果你能从他们那里请求、借、偷甚至是购买它，还是有意义的。在 AltaVista 上快速搜索将会有几十个结果。当然，Sun 的 Java 服务器已经有了，但是它相对于我们的需要来说太复杂了。

但是，我发现过多的例子用于教材、大学课程之类。检查一些以后，我确定比较喜欢 `richard5.net/projects/webster.php3`，该服务器名叫 Webster。我需要做的事情它并不是都能做，但它是一个好的开端。另外，作者允许你通过 GNU 公共许可证对它进行拷贝，这样可以毫不费力地重用它。

服务器的代码相当短，少于 150 行（如果不计空行和注释，甚至更少）。但是，服务器不能正确地处理 URL 编码问题（浏览器用来将特定字符转换成 16 进制值的处理过程）。这个不难安装。

配置服务器依赖于两个文件：`server.properties` 和 `mimetypes.properties`。服务器属性文件设置服务器的根目录、端口号以及缺省的服务器文件。MIME 类型属性文件设置文件扩展名与文档类型的关系（例如，`.txt` 属于 `text/plain` 类型，`.htm` 属于 `text/html` 类型）。

`server.properties` 文件可以包含下述任意关键字：

- `Portnumber`——服务器的端口号
- `Root`——文档的根目录
- `Defaultfile`——当浏览器请求一个目录时使用的缺省文件

下面是 `mimetypes.properties` 文件的一个简单示例：

```
htm=text/html
html=text/html
jpg=image/jpeg
jpeg=image/jpeg
gif=image/gif
```

当然，如果你需要，还可以添加更多的类型。

13.1.2.3 第二步：作些改变

为了将 Webster 程序用到服务当中，需要再做四件事：

- 正确处理 URL 编码串。
- 解析传到服务器上的查询串（例如，表单中使用 GET 操作的那部分）。
- 可以重载一个函数，执行解析文件名以及查询串以后、发送文件到客户端之前的工作。

- 处理解决问题的 HTTP 请求

第一个变化是很容易的，尽管 URL 对象有一个方法用于编码串，但它不做解码。但是对串进行扫描，把加号替换成空格和等值的 16 进制序列（也就是，16 进制数前边带上百分号）。你可以在清单 13.3 中找到 URLDecode 方法（和剩下的服务器代码一起）以及 HttpServer 类。

清单 13.3 HTTP 服务器的基类

```
// Small HTTP Server by Al Williams
// This code is based on the Webster server at
// http://richard5.net/projects/webster.php3
// which is covered by the GNU General Public License
// detailed at http://www.gnu.org/copyleft/gpl.html

import java.net.*;
import java.io.*;
import java.util.*;

public class HttpServer implements Runnable {

    private ServerSocket ss;
    boolean simple; // simple request?
    private Thread runner=null;

    // The server's configuration information is stored in these properties
    protected static Properties props = new Properties();

    // The mime types information is stored in this properties list
    protected static Properties MimeTypes = new Properties();

    HttpServer() { // main constructor
        runner = new Thread(this);
        runner.start();
    }

    public static void main(String[] args) {
        new HttpServer();
    }

    // override this to provide an action
    public String action(String filename, Hashtable vars) {
        System.out.println("Serving: "+filename + " " + vars);
        return filename;
    }

    // override this if you want to provide a raw file
    public DataInputStream openFile(String filename, Hashtable vars)
        throws IOException {
```

```
        return new DataInputStream(
            new BufferedInputStream (new
                FileInputStream(fullFileName(filename))));
    }

    public String fullFileName(String filename) {
        return HttpServer.props.getProperty("root") +
            filename;
    }

    public void run() {
        try{
            loadProps();
            loadMimes();
            System.out.println("HttpServer listening on port:" +
                props.getProperty("portnumber"));
            // setup serversocket
            ss = new ServerSocket( (new Integer(props.getProperty
                ("portnumber"))).intValue() );

            while(true) {
                Socket s = ss.accept(); // accept incoming requests
                new Thread(new SendFile(this,s)).start();
            }
        } catch(Exception e) {
            System.out.println("Main Serve thread ' + e );
        }
    }

    // load the properties file
    static void loadProps() throws IOException {
        File f = new File("server.properties");
        if (f.exists()) {
            InputStream is =
                new BufferedInputStream(new FileInputStream(f));
            props.load(is);
            is.close();
        }
    } // end of loadProps

    // load the properties file
    static void loadMimes() throws IOException {
        File f = new File("mimetypes.properties");
        if (f.exists()) {
            InputStream is =
                new BufferedInputStream(new FileInputStream(f));
            MimeTypes.load(is);
            is.close();
        }
    } // end of loadMimes
```



```
} // end of Serve class

class SendFile implements Runnable{
    private Socket client;
    private String fileName,header;
    private String query;
    private DataInputStream requestedFile;
    private int fileLength;
    private HttpServer svr;

    SendFile(HttpServer svr,Socket s) { // constructor
        client = s;
        this.svr=svr;
    }

    public void run() {
        String line;
        try {
            BufferedReader dis =
                new BufferedReader(new InputStreamReader
                                   (client.getInputStream()));
            // read request from browser and parse
            while((line=dis.readLine())!=null)
                StringTokenizer tokenizer = new StringTokenizer(line," ");
                if (!tokenizer.hasMoreTokens()) break;
                if (tokenizer.nextToken().equals("GET")) {
                    fileName = tokenizer.nextToken();
                    if (fileName.endsWith("/")) {
                        fileName = fileName +
                            HttpServer.props.getProperty("defaultfile");
                    } else {
                        fileName = fileName.substring(1);
                    }
                    String type;
                    try {
                        type=tokenizer.nextToken();
                    }
                    catch (NoSuchElementException nsee)
                    {
                        type=null;
                    }
                    svr.simple=type==null;
                    if (svr.simple) break;
                }
            }
        if (fileName.charAt(0)!='/') fileName = '/' + fileName;
        int n=fileName.indexOf('?');
```

```

if (n!=-1) {
    query=fileName.substring(n+1);
    fileName=fileName.substring(0,n);
}
else
    query="";
fileName=URLDecode(fileName);
// decode query string
Hashtable qvars = new Hashtable(64);
int n0,n1;
do {
    String val,key;
    n0=query.indexOf('&');
    if (n0==-1) n0=query.length();
    if (n0<=0) break;
    String vpart = query.substring(0,n0);
    if (n0==query.length()) query="";
    else query=query.substring(n0+1);
    n1=vpart.indexOf('=');
    if (n1==-1) {
        val="";
        key=vpart;
    } else {
        val = vpart.substring(n1+1);
        key = vpart.substring(0,n1);
    }
    qvars.put(URLDecode(key),URLDecode(val));
} while (!query.equals(""));
fileName=svr.action(fileName,qvars);
try {
    requestedFile=svr.openFile(fileName,qvars);
    fileLength = requestedFile.available();
    constructHeader();
} catch (IOException e) { // file not found send 404.
    header = "HTTP/1.0 404 File not found\n" +
        "Allow: GET\n" +
        "MIME-Version: 1.0\n"+
        "Server : HttpServer: a Java Local HTTP Server\n"+
        "\n\n <H1>404 File not Found</H1>\n";
    fileName = null;
}
int i;
DataOutputStream clientStream =
    new DataOutputStream(new BufferedOutputStream(client.
        getOutputStream()));
if (!svr.simple) clientStream.writeBytes(header);
if (fileName != null) {
    while((i = requestedFile.read()) != -1) {
        clientStream.writeByte(i);
    }
}

```

```

        }
    }
    clientStream.flush();
    clientStream.close();
    dis.close();
    client.close();
    if (requestedFile!=null) requestedFile.close();
} catch(Exception e) {
    System.out.print("Error closing Socket\n"+e);
}
}

public String URLDecode(String in)
{
    StringBuffer out = new StringBuffer(in.length());
    int i = 0;
    int j = 0;
    while (i < in.length())
    {
        char ch = in.charAt(i);  i++;
        if (ch == '+') ch = ' ';
        else if (ch == '%')
        {
            ch = (char) Integer.parseInt(
                in.substring(i,i+2), 16);
            i+=2;
        }
        out.append(ch);
        j++;
    }
    return new String(out);
}

private void constructHeader() {
    String fileType;
    fileType = fileName.substring(fileName.
        lastIndexOf(".")+1,fileName.length());
    fileType = HttpServer.MimeTypes.getProperty(fileType);
    header = "HTTP/1.0 200 OK\n" +
        "Allow: GET\nMIME-Version: 1.0\n"+
        "Server : HttpServer : a Java Local HTTP Server\n"+
        "Content-Type: " + fileType + "\n"+
        "Content-Length: " + fileLength +
        "\n\n";
}
}

```

使用哈希表来解析查询串看起来是一个很自然方法。Hashtable 对象（从实际使用来看）是一个相联数组。因此，给定一个 Hashtable 名为 vars，和一个 URL 值为：

```
http://host/somefile.htm?Name=A1&Member=Y
```

你就可以写如下的代码：

```
if (vars.get("Member").equals("Y")) memberpage();
```

在串中搜索 “&” 和 “=” 号都比较简单，URL 解码发生在解析工作之后，这样数据中的任何等号或者 “&” 号仍然保持编码状态，不会与解析程序产生混淆。

一旦程序使用 Hashtable，它调用 action 方法。action 方法需要两个参数：文件名以及带有查询串变量的一个 Hashtable。函数返回服务器要发送给客户端的文件名。文件名将确定文档的 MIME 类型（来自 mimetypes.properties 文件）。

清单 13.3 中的 action 函数简单地将两个参数的内容打印到 Java 控制台，并返回服务器传给它的相同的文件名。但是，扩展服务器类的那些类将会提供关于 action 的更多有意义的实现。

除了 action 方法，新代码提供了另一个可以重载的方法——openFile。该方法的参数与 action 相同。但返回值为 DataInputStream 类型。服务器使用 Stream 而不是 Reader，因为 Web 服务器不管怎样只返回字节值。

最后一个变化是使用新的 simple 变量，如果还有第三个标识用在请求行，服务器假设它是某个 HTTP 版本号（例如 HTTP/1.0），并设置 simple 值为 false。如果没有第三个标识符，服务器设置 simple 值为 true，并且不会等待任何标题。同时，如果 simple 值为 true，服务器不返回任何标题。

经过这些改变，该类成为一个有用的基类，可以让你将其扩充，做很少的工作就能定制服务器。

13.1.2.4 第三步：问题解决

如你所想，原来的问题是要将表单存进以逗号分隔的文件中。使用清单 13.3 中的 HttpServer 类以后，它就是一个很简单的任务了。在清单 13.4 中，可以发现关于 LocalServer 的代码，LocalServer 的代码就可以实现这个任务。

清单 13.4 这个本地服务器允许 Web 页面存储文件

```
import HttpServer;
import java.util.*;
import java.io.*;

public class LocalServer extends HttpServer {
    public static void main(String[] args) {
        new LocalServer();
    }

    public String action(String filename, Hashtable vars) {
        String realfile=filename;
        // here we see if there is file name query
        // variable--if there is, we assume
        // this is really the "CGI" file
        if (vars.containsKey("Filename")) {
```

```

        try {
            int items=0;
            FileWriter fw =
                new FileWriter((String)vars.get("Filename"),true);
            vars.remove("Filename");
            Enumeration e=vars.keys();
            while (e.hasMoreElements()) {
                if (items++!=0) fw.write(",");
                Object k=e.nextElement();
                fw.write("\""+k+"\"=\""+vars.get(k) + "\"");
            }
            fw.write('\n');
            fw.close();
        } catch (IOException e) {
            realfile="error.htm";
        }
    }
    else
        realfile="/response.htm";
    return realfile;
}
}

```

LocalServer 类包含一个 **main** 例程（这样它可以执行）以及重载的 **action** 方法。执行上述简单任务所需要的就是这些。**action** 例程假设带有 **Filename** 变量的任何页是一个伪脚本。它使用下述代码来检查这个变量：

```
if (vars.containsKey("Filename")) {
```

关键字 **Filename** 区分大小写。一旦程序发现这个变量，它就使用该变量值作为追加内容的文件名。它也从 **Hashtable** 里移去该变量值，因此程序将不会把这个内部变量写到文件里。

使用 **FileWriter** 类打开该文件比较简单。它的构造函数带两个参数，分别为文件名以及一个表示追加标志的布尔变量。一旦准备好对象，对 **write** 的简单调用将发出我们需要的数据。调用 **close** 将关闭文件。

因为我的目标是支持使用这种服务器的单个用户，没有特别注意防止多个用户同时试图对相同的文件进行写操作。但是，这不难添加，将 **action** 方法同步就可以实现（尽管牺牲了部分性能）。

一旦程序打开文件，它就枚举 **Hashtable** 中的每个关键字（记住，**Filename** 关键字已经没有了）。接着为那些变量写一行，如果 **URL** 是：

```
http://localhost/anything.cgi?Name=A1&Extension=32
```

程序将执行如下写操作：

```
"Name"="A1","Extension"="32"
```

我没有考虑数据中带有引号的情况，但是可以很简单地将引号替换成 **** 或 **""**。以让程序读文件。

13.1.2.5 超出本地服务器

这个简化的例子有大量的东西可以扩展,使用 `LocalServer` 可以创建任意数目的带有 `action` 方法的继承类。当然,你可以每次只在一个服务器的一个端口上运行,也可以在 `server.properties` 文件中设置不同的端口。这样,如果端口号是 88,就可以使用 URL: `http://localhost:88/foo.txt`。

下面是一些关于使用 `HttpServer` 类的思想:

当访问一个特定文件的请求到达时,你可以在另一个源(如网络或者串行端口)上读取数据,并发送数据。这将让你显示实时网络状态、传感器上的数据等。

可以在这个思想上创建服务器,使用 `action` 调用 Perl 或者是 shell(使用 `Runtime` 对象的 `exec` 方法)。接着你可以写真正的 Perl CGI 脚本。这将使得服务器更像一个完整的 Web 服务器,但是从某种角度来看,你可能也要使用一个完整的 Web 服务器。

有可能还要让服务器检查是否存在真实网络服务器,如果发现了真实地服务器,Java 程序可以将 HTTP 重定向,强制浏览器进入真实网页。

要求 `action` 例程打开一个字节流,送往本地服务器将是一个好特征。因为该字节流不一定是一个文件,它可能会让 `action` 例程发送客户化的数据而不经中间文件。

一旦你创建了一个简单的 Web 服务器,将发现自己将 Web 浏览器更多地看作是 Java 程序的前端。通过组合 HTML 和 Java,你将会有新的方法来安排程序,使它能以可移动的方式在 Web 上运行或者在本地运行。

如果你知道 Web 服务器是如何工作的,写一个这样的程序就不难了。由于在网上能找到免费的代码,创建这个应用就容易得多了。不要忘记在启动任何工程之前,查一下网上的资料,可能会节省一部分工作量。即使你不使用直接发现的代码,研究代码也会对你自己写程序有帮助。

小型服务器有很多用户,Java 使得编写它更为容易。我在这里提供的类使得使用 Java 面向对象更为容易。你只需要简单地对 `HttpServer` 类进行扩展,并提供你需要的指定内容即可。

13.1.3 通过代理创建的 Web

现在的工作地点比过去更容易引起争论。如果你暗中窥视雇员的秘密,你有可能被起诉。同时,如果一个雇员向另一个人泄露有争论的材料,也有可能被起诉。除了法律问题外,监视雇员会引起若干有争议的道德问题。这看起来就像是没有胜出的情形。

尽管你不能控制雇员见到的所有内容,你可以使用代理服务器来控制他们工作时访问的 Internet 站点。例如,代理服务器可以中断对特定类型的网站内容的访问,甚至阻止对特定网站的访问。

但是合法的监视只是冰山一角。实质上,一个代理服务器只是充当浏览器与 Web 服务器之间的一个中介,用于处理你对网上通路的请求。代理服务器让你执行有关浏览器请求所有类型的处理,这些处理是良性的,同时控制也比较严格。它可以过滤掉像广告、参照者字符串、cookie,或者预取和缓存网页,以让拨号连接更快一些。代理服务器也可以管制吞吐量

并跟踪网络访问。

13.1.3.1 基础

不管你决定如何使用它们，代理服务器监视 HTTP 的通信过程如下：

1. 内部的 Web 浏览器发送一个请求到代理服务器，它驻留在防火墙或者网关计算机上，请求的第一行包含了要访问的 URL。
2. 代理服务器读 URL，并获取 URL，将请求传给正确的终点计算机。
3. 代理服务器接收来自 Internet 上的终点计算机的响应，并将它路由到正确的内部 Web 浏览器。

例如，假设一个职员企图访问一个应用网站，如果没有代理服务器，职员的浏览器向包含那个站点的 Web 服务器打开一个套接字，数据从 Web 服务器直接传输给职员的浏览器。但是，如果浏览器设置使用代理服务器，请求传送到代理服务器。那么，代理服务器获得请求第一行中的 URL，并向 WebTechniques.com Web 服务器打开一个套接字。当数据从 Web 服务器返回时，代理服务器将它路由到职员的浏览器。

有些 HTTP 代理读标题文件来控制像缓存 (cache) 这样的处理，例如，代理可能存储 (缓存) 网页，这样它可以满足将来的请求，而不用访问远程的服务器。在这种情况下，代理必须考虑控制缓存过期的标题以及缓存控制。简单的代理不需要对任何标题进行响应。

13.1.3.2 不仅仅适用于公司

当然，代理服务器不只是适用于公司。作为一个开发人员，我发现拥有自己的代理服务器是很有用的。因为它让我检查浏览器到 Web 服务器之间的通路。当在 Web 应用中遇到问题时，这就成了关键。你甚至可以使用多个代理服务器 (大多数代理让你将多个链接在一起)。例如，一个可能是公司的代理服务器，另一个可能是基于 Java 的代理服务器，让你收集调试信息的。但要记住，链中的每个代理在性能上将会有更多的损失。

13.1.3.3 只是一个服务器

像名字所指的一样，代理服务器实际上只是一个专门的服务器。像大多数服务器一样，如果你想用它来处理多个请求，那就需要使用多线程。下面是基本思路：

1. 等待来自一个客户端的连接 (一个 Web 浏览器)。
2. 启动一个新线程来处理连接。
3. 读浏览器请求的第一行 (包含终点 URL 的行)。
4. 解析请求第一行中终点 Web 主机名及其端口号。
5. 打开一个到终点 Web 主机的套接字或者是上传流的代理服务器 (如果可行的话)。
6. 向外发套接字发送请求的第一行。
7. 向外发套接字发送请求的剩余部分。
8. 发送从终点 Web 主机 (通过套接字) 返回给浏览器请求的数据。

当然，其中的细节要复杂一点。实质上，有两个重要的障碍要克服：第一，一般更倾向于从套接字中逐行读取数据，但那样会引起性能上的瓶颈。第二，你需要一个高效方法来连

接两个套接字。有若干方法可以实现这两个目标，每种方法都有利弊。例如，如果在数据到达时对其进行过滤，逐行读取数据可能是最好的选择。但是，在大多数情况下，最好还是在请求到达代理服务器时立即发送数据。同时，你可以使用独立的线程来发送和接收数据，但是，创建和释放大量的线程可能引起性能上的问题。所以，我决定使用一个线程处理对每个请求的发送和接收，并试图在请求到达代理时立即发送数据。

13.1.3.4 示例

使用基于 Java、面向对象的可重用的代理服务器是一种好思想。使用这种方法，你可以将它用于其他项目，在那些项目中你需要对浏览器请求作不同的处理。当然，你需要在灵活性和效率之间作出权衡。为了创建自己的代理服务器，我通过扩展 Thread 基类来创建 HttpProxy 类（见清单 13.5 中的源代码）。该类包括能对某些代理服务器行为进行定制的属性（见后边的“写一个代理服务器”一节中的表 13.1）。

清单 13.5 该类可以充当一个代理服务器或者为定制代理提供一个基类

```

/*****
 * Proxy Server Base Class - Williams
 *****/
*/
import java.net.*;
import java.io.*;

public class HttpProxy extends Thread {
    static public int CONNECT_RETRIES=5;
    static public int CONNECT_PAUSE=5;
    static public int TIMEOUT=50;
    static public int BUFSIZ=1024;
    static public boolean logging = false;
    static public OutputStream log=null;
    // inbound
    protected Socket socket;
    // optional parent proxy
    static private String parent=null;
    static private int parentPort=-1;
    static public void setParentProxy(String name, int pport) {
        parent=name;
        parentPort=pport;
    }

    // Create a proxy thread on a given socket
    public HttpProxy(Socket s) { socket=s; start(); }

    public void writeLog(int c, boolean browser) throws IOException {
        log.write(c);
    }
}

```

```
public void writeLog(byte[] bytes,int offset, int len,
    boolean browser) throws IOException {
    for (int i=0;i<len;i++) writeLog((int)bytes[offset+i],browser);
}
```

```
// Subclasses may override
// By default, just log to stdout
public String processHostName(String url, String host, int port,
    Socket sock) {
    java.text.DateFormat cal=
        java.text.DateFormat.getDateTimeInstance();
    System.out.println(cal.format(new java.util.Date()) + " - " +
        url + " " + sock.getInetAddress()+"<BR>");
    return host;
}
```

```
// Here is the thread that does the work
public void run() {
    String line;
    String host;
    int port=80;
    Socket outbound=null;
    try {
        socket.setSoTimeout(TIMEOUT);
        InputStream is=socket.getInputStream();
        OutputStream os=null;
        try {
            // get request line
            line="";
            host="";
            int state=0;
            boolean space;
            while (true) {
                int c=is.read();
                if (c==-1) break;
                if (logging) writeLog(c,true);
                space=Character.isWhitespace((char)c);
                switch (state) {
                    case 0:
                        if (space) continue;
                        state=1;
                    case 1:
                        if (space) {
                            state=2;

```

```

        continue;
    }
    line=line+(char)c;
    break;
case 2:
    if (space) continue; // skip multiple spaces
    state=3;
case 3:
    if (space) {
        state=4; // doesn't really matter
        // isolate just host name
        String host0=host;
        int n;
        n=host.indexOf("//");
        if (n!=-1) host=host.substring(n+2);
        n=host.indexOf('/');
        if (n!=-1) host=host.substring(0,n);
        // need to parse possible port from host
        n=host.indexOf(":");
        if (n!=-1)
            port=Integer.parseInt(host.substring(n+1));
        host=host.substring(0,n);
    }
    host=processHostName(host0,host,port,socket);
    if (parent!=null) {
        host=parent;
        port=parentPort;
    }
    int retry=CONNECT_RETRIES;
    while (retry--!=0) {
        try {
            outbound=new Socket(host,port);
            break;
        } catch (Exception e) { }
        // wait
        Thread.sleep(CONNECT_PAUSE);
    }
    if (outbound==null) break;
    outbound.setSoTimeout(TIMEOUT);
    os=outbound.getOutputStream();
    os.write(line.getBytes());
    os.write(' ');
    os.write(host0.getBytes());
    os.write(' ');
    pipe(is,outbound.getInputStream(),os,socket.getOutputStream()
eam());

    break;
}

```



```

        host=host+(char)c;
        break;
    }
}
}
catch (IOException e) { }

} catch (Exception e) { }
finally {
    try { socket.close(); } catch (Exception e1) {}
    try { outbound.close(); } catch (Exception e2) {}
}
}

void pipe(InputStream is0, InputStream is1,
    OutputStream os0, OutputStream os1) throws IOException {
    try {
        int ir;
        byte bytes[]=new byte[BUFSIZ];
        while (true) {
            try {
                if ((ir=is0.read(bytes))>0) {
                    os0.write(bytes,0,ir);
                    if (logging) writeLog(bytes,0,ir,true);
                }
                else if (ir<0)
                    break;
            } catch (InterruptedException e) { }
            try {
                if ((ir=is1.read(bytes))>0) {
                    os1.write(bytes,0,ir);
                    if (logging) writeLog(bytes,0,ir,false);
                }
                else if (ir<0)
                    break;
            } catch (InterruptedException e) { }
        }
    } catch (Exception e0) {
        System.out.println("Pipe Exception: " + e0);
    }
}

static public void startProxy(int port,Class clobj) {
    ServerSocket ssock;
    Socket sock;
    try {

```

```

        ssock=new ServerSocket(port);
        while (true) {
            Class [] sarg = new Class[1];
            Object [] arg= new Object[1];
            sarg[0]=Socket.class;
            try {
                java.lang.reflect.Constructor cons =
                    clobj.getDeclaredConstructor(sarg);
                arg[0]=ssock.accept();
                // create new HttpProxy or subclass
                cons.newInstance(arg);
            } catch (Exception e)
                Socket esock = (Socket)arg[0];
                try { esock.close(); } catch (Exception ec) {}
            }
        } catch (IOException e) {}
    }
    // if we return something is wrong!
}

// Very simple test main
static public void main(String args[]) {
    System.out.println("Starting proxy on port 808<BR>");
    HttpProxy.log=System.out;
    HttpProxy.logging=true;
    HttpProxy.startProxy(808,HttpProxy.class);
}
}

```

当代理服务器连到 Web 主机之后, 我使用一个简单的循环在套接字之间中转数据。一个潜在的问题是如果数据不可得, 那么对 read 方法的调用可能引起程序的中断, 从而导致程序挂起。为防止产生这个问题, 我使用了 setSoTimeout 方法来设置套接字是否超时(见清单 13.5)。这样, 如果一个套接字处于非活动状态, 另外一个还有机会处理, 同时我不必创建一个新线程。对于大量的数据, 创建另一个线程可能更高效。但是, 对于数据量小些的情形, 创建新线程会浪费过多的资源, 从而使潜在性能下降。

清单 13.5 也显示了非常简单的 main 方法, 可以用来对 HttpProxy 类进行测试。大量的工作发生在方法 startProxy 那里。该方法使用了一个不常用的技术, 以允许它的一个静态成员来创建 HttpProxy 类的一个实例(或者它的一个子类)。其思想是传递一个 Class 对象给 startProxy 方法。接着, startProxy 方法通过反射(最新版 Java 的一种工具)方法来确定那个 Class 对象接收一个 Socket 作为参数, 使用哪个构造函数。最后, startProxy 方法调用 newInstance 方法来创建该对象的一个实例。

这个技术让你扩展 HttpProxy 类, 而不用创建 startProxy 方法的一个客户化版本。为了找

到一个特殊类型的 Class 对象，简单地将.class 附加在普通名字之后。(如果有一个对象的实例，就调用 getClass 方法。) 因为将 Class 对象传给 startProxy 方法，所以在创建 HttpProxy 的子类时不必作特殊的改变。(后边的清单 13.4 显示了一个简单的代理服务器子类的例子。)

13.1.3.5 线程

像所有线程对象一样，HttpProxy 类在 run 方法里实现它的主要任务。run 方法实现一个简单的状态机，用于每次从 Web 浏览器中读字符。一直继续到收集完足够的信息，找到目标 Web 服务器为止。接着，run 方法打开一个面向这个 Web 服务器的套接字。(如果有多个代理串在一起，方法就打开面向这个长串中下一个代理服务器的套接字)。

在套接字打开以后，run 方法发送部分请求到套接字并调用 pipe 方法。pipe 方法只是在两个套接字之间使用最小化的接口进行读和写操作。

13.1.3.6 定制

使用子类来定制或者调节代理服务器的行为有两种方法，主要是通过改变主机名或者捕获流经代理的所有数据这两种方法。processHostName 方法让代理检查并改变主机名。如果允许进行日志记录，代理服务器可以为流经代理的每个字符调用 writeLog 方法。

怎样处理那些信息，就在于你自己，程序可以将数据写进日志文件，将它发送到控制台或者做适合于你的目标的任何事。在 writeLog 输出的一个布尔标志指出它是否来自一个 Web 浏览器或者一个 Web 主机。

13.1.3.7 下一步将是什么

像大多数工具一样，代理服务器本来没有好坏之分。它的好坏依赖于你如何去使用它。一个代理可以侵犯隐私，但是它也可以保护你的网络免受入侵，也保护你免受诉讼。

我喜欢将代理服务器看作是扩展 Web 浏览器的一种方式，即使它不在同一台计算机上。例如，你在发送数据到浏览器之前可以使用代理来压缩数据。一个非常复杂的代理甚至可以将一种语言翻译成另一种语言，在经过它的时候可以管理 accept 标题(题头)。

13.1.4 拍卖服务器

像很多人一样，我曾经因为在 eBay 上进行买卖而闻名。近来，我的母亲成了网络的新客人，想从拍卖站点买一些物品，因此我努力向她解释在线的进出交易。对新手最难解释的一件事是人们如何“诽谤”拍卖，他们一直等待，直到最后一分钟没有人出高价购买那些商品为止。当然，在传统的拍卖中，这是不可能的，拍卖不会停止，直到每个人都有机会出价为止。

有些其他的拍卖站点通过自动扩展拍卖的终止期限来传播信息，直到投标结束。但是，像我的儿子 Patrick 指出的一样，它还不能捕获拍卖的实况，我决定创建一个活动的拍卖环境，在这个环境当中，买者实时互相进行竞价，一次拍卖直到每个人停止出价时才结束。

因为实时拍卖中的数据是不断发生变化的，系统要求在出价的小程序(applet)和拍卖服务器之间有一个持续的连接。为让这个连接成为可能，我创建了一个定制的 Java 服务器，与

运行在每个买者的 Web 浏览器上的出价（投标）applet 进行交互。每次拍卖需要一个独立的服务器来创建一个套接字来监听连接。那意味着每次拍卖处理将不得不使用独立的端口号。

有关拍卖的大多数信息是静态的（说明、图像、起始投标），因此我决定使用一个 JSP 脚本来显示该内容。一个简单的静态 HTML 是不够的，因为每次拍卖产生的页面是不同的。并且使用 applet 显示这些信息工作量更大，因为你不得不对所有的数据进行格式化和排版。这样系统分成若干部分：

- 定制的服务器
- 一个 applet
- 置于标准的 Web 服务器上的一个 JSP 页面
- 带有拍卖数据的数据库

13.1.4.1 穷人的数据库

尽管系统使用 Java 服务器的三个完全不同的部分：Java 服务器、applet 和 JSP 页。它们都能令人满意地工作。那意味着系统将需要某种方法来保持这三个部分的同步。理想情况下，拍卖数据驻留在数据库中，而数据库对于三个部分都是可访问的。但是，我想让工作变得更简单，于是我选择使用 Java 属性文件。为此，所有数据库中你都需要它。当然，applet 不能读 Web 服务器上的属性文件，但是因为它不得不与制定的服务器通信，我已经设计了相应的程序，它可以通过第二种通道获取它需要的数据。

属性文件的一个优点是使用普通的文本编辑器很容易对它们进行创建。在 servlet(JSP 文件要转变成的文件，或者从一个用于服务器的普通的 Java 程序中转变得来)里访问属性文件也简单。如果你想对拍卖结果进行标识，那么，使用这两种技术，可以回写一个属性文件到服务器。

要访问一个属性文件，可以创建一个 InputStream 对象（例如，使用 FileInputStream）并将它传给 Properties 对象的 load 方法。记住在一个 JSP 文件里，打开的是相对于服务器的根目录的文件。其中的技巧是使用 getServletContext 方法来获得一个 ServletContext 对象。接着你可以调用 getResourceAsStream 方法来将相对于服务器的 URL 转变成一个输入流。

清单 13.6 显示了一个拍卖的属性文件。“#”号表示一个注释，这样下面的文本并不是数据库的一部分。Description 属性既可以是文件，也可以是包含描述的相对 URL。如果属性以斜线开头，服务器假设它是一个 URL，否则，它将属性看作是文本。

清单 13.6 该属性文件定义了一次拍卖

```

IMG=http://www.al-williams.com/wd5gnr/tivo.jpg      # Item image
port=7542                                           # Server port
Description=/auction/tivo.htm                      # Description
StartBid=11.00                                     # Start bid
EndAt=30                                            # Duration (minutes)
title=Phillips\ TiVo\ Digital\ Recorder           # Title
Short=Phillips\ TiVo                               # Short description
  
```

13.1.4.2 一个 Java 拍卖服务器

整个服务器的代码显示在清单 13.7 当中。因为要从命令行中启动服务器,我在静态的 main 方法里实现了它。该函数的大多数代码对变量进行了初始化。

清单 13.7 拍卖服务器在 JSP 与定制的 applet 的联合体中运行

```
// Auction Server

import java.io.*;
import java.util.*;
import java.net.*;

public class AuctServer extends Thread
{
    static String ulist;

    Socket csock;           // the socket
    static Calendar endtime; // end time of auction
    static float highbid=0.0f; // high bid
    static String bidder;    // high bidder
    static boolean done=false; // finished?
    static String pfn;       // property file name
    static Properties p;     // property database

    public AuctServer(Socket skt) { csock=skt; }

    // handle a bid
    synchronized private void bid(String bidid, float bidamt)
    // synchronized
    {
        Calendar cal=Calendar.getInstance();
        long diff;
        System.out.println(bidid + " bids " + bidamt);
        // check for time
        if (cal.after(endtime)) return; // too late
        // check for bid amount
        if (highbid>=bidamt) return;
        // if all OK set high bid
        System.out.println("High bid: " + bidid + " = " + bidamt);
        highbid=bidamt;
        bidder=bidid;
        // and adjust time if <60 seconds left
        diff=(endtime.getTime().getTime()-cal.getTime().getTime())/1000;
        if (diff<60) endtime.add(Calendar.SECOND,60);
        // add 1 more minute
    }

    // generate status
```



```

synchronized private void status(PrintWriter pw)
{
    Calendar cal=Calendar.getInstance();
    long diff;
    // check time
    diff=(endtime.getTime().getTime()-cal.getTime().getTime())/1000;
    if (diff<=0 && !done)
    { // if first time we've noticed we are done, complete auction
    done=true;
    p.put("HighBid",Float.toString(highbid));
    p.put("HighBidder",bidder);
    System.out.println("Auction complete "+ bidder
        + " " + highbid);
    try
    {
        p.store(new FileOutputStream(pfn),"");
        // p.save is deprecated!
    } catch (Exception e)
    {
        System.out.println("Error ending auction");
        System.out.println(e.getMessage());
        e.printStackTrace(System.out);
    }
    }
    if (diff<0) diff=0; // don't return negative time
    pw.print(diff);
    pw.print(":");
    pw.print(highbid);
    pw.print(":");
    pw.print(bidder);
    pw.print("\n");
    pw.flush();
}

// each client thread runs this code
public void run()
{
    String cmd;
    try
    {
        InputStream istream = csock.getInputStream();
        PrintWriter pw = new PrintWriter(csock.getOutputStream());

        OutputStream ostream = csock.getOutputStream();
        BufferedReader br = new
            BufferedReader(new InputStreamReader(istream));
    }
    // read commands
    while (true) {

```

```

        cmd=br.readLine();
        if (cmd==null) break;    // connection dropped?
        if (cmd.charAt(0)=='S') // status command
        {
            status(pw);
        }
        if (cmd.charAt(0)=='B') // bid
        {
            String id;
            float amt;
            int n;
            n=cmd.indexOf(':');
            id=cmd.substring(1,n);
            amt=Float.valueOf(cmd.substring(n+1)).floatValue();
            bid(id,amt); // parse these out
            status(pw);
        }
    }
} catch (Exception e) { }
}

// This is the server's main entry point
public static void main(String args[]) throws Exception
{
    ServerSocket sock;
    p=new Properties();
    pfn="auction" + args[0] + ".properties";
    p.load(new FileInputStream(pfn)); // read database

    // compute end time
    endtime=Calendar.getInstance();
    endtime.add(Calendar.MINUTE,
        Integer.parseInt(p.getProperty("EndAt")));

    // create server socket
    sock=new ServerSocket(Integer.parseInt(p.getProperty("port")));

    // accept connections from clients
    while (true) {
        new AuctServer(sock.accept()).start();
    }
}
}

```

AuctServer 类扩展了 **Thread** 类。这样当这个静态函数创建一个新的实例时，它实际上启动了一个新的执行线程。静态的成员变量允许所有的实例进行通信。

客户端（那些 **applet**）可以发送两种命令。所有的命令都以新行符结束。**S** 命令不带参数，并促使服务器回送一个状态串。**B** 命令放入一个投标。紧跟 **B** 命令的是用户的 ID、一个

冒号以及出价的多少。这个命令也回送当前的拍卖状态。

拍卖状态是一个字符串，客户端可以使用它来显示拍卖的状态。这个串是包含有拍卖维持的时间秒数、一个冒号、高出价、一个冒号以及高出价者的 ID。状态串也以新行符结束。

因为 bid 方法和 status 方法更改共享变量，它们需要对属性进行同步。这阻止了多个线程同时访问它们，从而混淆了拍卖的状态。

服务器并不接受拍卖结束以后的出价，或者有可能出价比当前价格要低（或者相同）。但是，如果离拍卖结束不到一分钟内还有人出价成功，那么结束时间就增加一分钟。这与现实拍卖中的“第一次出价、第二次出价...”很类似。它确保了每人都有机会出价。

status 命令不会报告负的持续时间。当状态读到剩余时间为 0 秒时，拍卖就结束了。在拍卖结束时，第一个状态请求会促使服务器打印该拍卖的结果，并将结果保存到属性文件当中。

服务器要求你在命令行指定一个串，用于查找正确的属性文件。例如，如果你在命令行里放入“1”，服务器使用的就是 auction1.properties 属性文件。因为服务器的端口号存在属性文件里，所以可以让你同时运行多个拍卖服务。

13.1.4.3 测试服务器

使用独立的程序片来写一个系统的问题之一是你需要使用所有的片段来做任意的真实测试。但是，如果像我一样，第一次就将每个片段都写正确的机率很小。

为了除去早期的 bug，我常常写些小测试程序来检查部分程序。因为这些测试程序写起来快而且简单易写，你可以在进行下一个更大更重要的模块之前，使用它们来对更大的程序进行排错。我写了两个非常简单的程序用于这个目的：t.java（清单 13.8）模拟一次出价，ts.java（清单 13.9）请求服务器的状态。这些程序伴随着服务器中的一些临时的 System.out.println 语句，允许我将 AuctServer 对象启动运行。

清单 13.8 该程序向服务器发送测试出价

```
import java.io.*;
import java.util.Properties;
import java.net.*;

public class t
{
    static public void main(String args[]) throws Exception
    {
        Socket sock = new Socket("127.0.0.1",7542);
        PrintWriter pw=new PrintWriter(sock.getOutputStream());
        InputStream istream = sock.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(istream));
        String response;
        pw.print("S\n");
        pw.flush();
        response=br.readLine();
    }
}
```

```

        if (response==null) response="Connection terminated";
        System.out.println(response);
        br.close ();

    }
}

```

清单 13.9 该测试程序查询服务器的状态

```

import java.io.*;
import java.util.Properties;
import java.net.*;

public class ts
{
    static public void main(String args[]) throws Exception
    {
        Socket sock = new Socket("127.0.0.1",7542);
        PrintWriter pw=new PrintWriter(sock.getOutputStream());
        InputStream istream = sock.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(istream));
        String response;

        pw.print("S\n");
        pw.flush();

        response=br.readLine();
        if (response==null) response="Connection terminated";
        System.out.println(response);
        pw.close();
        br.close();
    }
}

```

13.1.4.4 创建 Applet

如果给定服务器的规范，applet 几乎可以自己写。这里的 applet 称作 client.java（见清单 13.10），使用了下述几个参数：

- id——用户的 email。
- port——服务器使用的端口。
- bgcolor——用于 applet 背景的 16 进制的颜色值。

applet 也使用线程，因为它每秒钟都轮询服务器检查拍卖是否处于活动状态。当拍卖的持续期为 0，applet 停止创建状态请求。这有助于减轻服务器的总体负荷。

清单 13.10 拍卖系统除了使用 JSP 和定制服务器以外，也使用该 applet

```
// Auction Applet
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

// parameters
// id = user's email or other id
// port = auction socket
// bgcolor = background color (hex)

public class AuctionApplet extends
    Applet implements Runnable, ActionListener
{
    // User interface elements
    Label hb= new Label("High Bid: ");
    Label hbval = new Label("00.00");
    Label hbidder = new Label("by:");
    Label hbidderval = new Label("                ");
    Label bidlbl = new Label("Your bid: $");
    TextField bid = new TextField(" 1.00");
    Button bidnow=new Button("Bid");
    Label tleft = new Label("Time Left:");
    Label tleftval = new Label("?????");

    // Sockets and streams
    Socket sock;
    PrintWriter pw;
    BufferedReader br;
    String response;

    boolean auctionactive=true;

    // helper: get parameter with default value
    private String getDefParam(String name,String def)
    {
        String rv=getParameter(name);
        if (rv==null) rv=def;
        return rv;
    }

    // helper: get integer parameter (with default value)
    private int getIntParam(String name,int radix,int defv)
    {
        int rv=defv;
        String p=getParameter(name);
        if (p!=null)
```



```

        try {
            rv=Integer.parseInt(p,radix);
        }
        catch (Exception e) { }
        return rv;
    }

// parse status response
private synchronized void status()
{
    try {
        response=br.readLine();
    }
    catch (Exception e) { response=null; }
    if (response==null) {
        tleftval.setText("Error");
    }
    else {
        int seconds;
        String timer;
        float highbid;
        String highbidder;
        String sep;
        int n,n1;
        java.text.NumberFormat fmt=
            java.text.NumberFormat.getCurrencyInstance();
        n=response.indexOf(':');
        seconds=Integer.parseInt(response.substring(0,n));
        if (seconds==0) auctionactive=false;
        n1=response.indexOf(':',n+1);
        highbid=Float.valueOf(response.substring(n+1,n1)).floatValue();
        highbidder=response.substring(n1+1);
        sep=":";
        if (seconds%60<10) sep=sep+"0";
        timer = Integer.toString(seconds/60)
            +sep+Integer.toString(seconds%60);
        tleftval.setText(timer);
        hbval.setText(fmt.format(highbid));
        hbidderval.setText(highbidder);
    }
}

// get started
public void init()
{
    setLayout(new FlowLayout(FlowLayout.LEFT));
    setBackground(new Color(getIntParam("bgcolor",16,0xFFFFFFFF));
// add elements

```

```
add(hb);
add(hbval);
add(hbidder);
add(hbidderval);
add(bidlbl);
add(bid);
add(bidnow);
add(tleft);
add(tleftval);
bidnow.addActionListener(this);
bid.setText(getDefParam("StartBid","1.00"));
// open up socket
try
{
// Important: you must run this via
// http://somehost to get the following to work
// if you are running from a file, you should
// hard-code your machine name here!
sock = new Socket(getDocumentBase().getHost(),
    getIntParam("port",10,0));
pw=new PrintWriter(sock.getOutputStream());
br=new BufferedReader(new
    InputStreamReader(sock.getInputStream()));
    new Thread(this).start(); // start status update thread
}
catch (Exception e)
{
    tleftval.setText("Net Error");
}
}

// button pushed?
public void actionPerformed(ActionEvent e)
{
    if (auctionactive)
    {
// make bid!
        pw.print("B"+getParameter("ID")+":"+bid.getText()+"\n");
        status();
    }
}

// thread to wait for 1 second and ask for status
public void run()
{
    while (auctionactive)
    {
        pw.print("S\n");
    }
}
```

```

        pw.flush();
        status();
        try {
            Thread.sleep(1000);
        } catch (Exception e) { }
    }
    try
    {
        sock.close();
    }
    catch (Exception e) { }
}

public void destroy()
{
    try
    {
        sock.close();
    }
    catch (Exception e) { }
}
}

```

服务器通过扩展 `Thread` 类来实现多线程。这不是 `applet` 的一个选择，因为它不得不扩展 `Applet` 类。因此，`applet` 实现 `Runnable` 类（换句话说，它有一个 `run` 方法），并创建一个新的 `Thread` 对象来引用该 `applet`。

`Applet` 的线程在 1 秒的时间内主要为睡眠状态，接着发出一个状态请求。如果用户按下了出价按钮，`applet` 就发送一个 `bid` 命令到服务器。

我故意将 `applet` 写成像一个条状，这样它更适合于拍卖网页的底端。当然，你可以将整个网页做成一个 `applet`，但是那会引发创建问题。例如，它很难显示 `applet` 中 HTML 格式的文本。在一个 JSP 页里，这是容易实现的事。

13.1.4.5 编写 JSP 脚本

在实际生活的系统中，你需要有一个机制来让你的出价者创建用户名，并对其进行某种方式的校验。例如，我简单地收集了出价者的姓名和邮箱（在清单 13.11 的 `index.jsp` 中），并假设这些信息是正确的。它对于显示当前拍卖列表也很有用，但是对本例而言，登录总是使用一个拍卖 ID，它的值为 1。

清单 13.11 用户使用这个主 JSP 来访问拍卖系统

```

<%@ page import="java.io.*" %>
<%@ page import="java.util.Properties" %>

<%!

```

```

String propfile;

String name;
String email;
Properties p;
InputStream str;
ServletContext ctx;
%>

<%
    propfile="/auction/auction";
    p=new Properties();
    ctx=getServletContext();
    name=request.getParameter("UName");
    email=request.getParameter("Email");

    if (name==null || name.equals("") || email==null || email.equals("")) {
%>
        <jsp:forward page="index.jsp" >
            <jsp:param name="err" value="Please enter a valid name and e-mail
address."/>
        </jsp:forward>

<%
    }
    propfile=propfile + request.getParameter("ANO") + ".properties";
    str=ctx.getResourceAsStream(propfile);
    if (str==null)
        out.println("Internal server error:" + propfile );
    else
    {
        p.load(str);
        str.close();
    }

%>
<HTML>
<HEAD><TITLE>Live auction</TITLE></HEAD>
<BODY BGCOLOR=WHITE>
<H1>
    <%= p.getProperty("Short") %>
</H1>
<P ALIGN=CENTER><IMG SRC=<%= p.getProperty("IMG") %>></P>
<P><FONT SIZE=+1>
<%
    String desc;
    desc=p.getProperty("Description");

```

```

    if (desc.charAt(0)=='/')
    {
        int c;
        InputStream is=ctx.getResourceAsStream(desc);
        try
        {
            while ((c=is.read())!=-1) {
                out.print((char)c);
            }
        }
        catch (Exception e) { }
    }
    else
        out.println(desc);
%>
<HR>
<P>Starting bid: $<%= p.getProperty("StartBid"); %>
</P>
<!-- start bidding applet -->
<APPLET code=AuctionApplet.class Width=640 Height=50>
<PARAM name=port Value=<%= p.getProperty("port") %>>
<PARAM name=bgcolor value=FFFFFF>
<PARAM name=ID value='<%= email %>'>
<PARAM name=StartBid value='<%= p.getProperty("StartBid"); %> >
</APPLET>
</BODY>
</HTML>

```

主拍卖页需要带一个参数（ANO 变量）来判断显示哪一个拍卖。该页确认存在一个名字和邮件地址。接着，它打开属性文件，读取有关拍卖的信息。剩余的页面只是对信息进行格式化。唯一的技巧部分是属性描述是以斜线开头，这表明它是一个 URL，而不是一个文本串。另外，`getResourceAsStream` 方法将 URL 转换成一个 `InputStream` 对象。从那里，可以很简单地将字符从流里拷贝到 Web 浏览器中。

13.1.4.6 综合

当然，`applet` 是在网页中完成实际的工作的。注意在由 JSP 脚本产生的页面结果中，`<APPLET>` 标签会促使浏览器装载 `applet`，脚本会将若干 `applet` 的属性动态地写进 HTML 页面。为测试系统自身，你需要进行如下操作：

1. 创建一个属性文件，名为 `auctionXX.properties`。
2. 使用类似于 `java AuctServer XX` 的命令行来启动服务器。必须与 Web 服务器相同的机器上启动拍卖服务器，因为为了安全的原因，`applet` 可能只与那台机器相连接。
3. 在 `index.jsp` 中将参数 `ANO=1` 改为 `ANO=XX`。
4. 一旦运行以后，使用 Web 浏览器来浏览 `index.jsp` 文件。（我使用的是 IIS5.0 和 Allaire 的 JRun3.0。）

5. 登录进去, 你可能需要从若干个浏览器登录进去, 或者从相同的机器, 或者从不同的机器, 立即模拟若干个出价者的行为。

13.1.4.7 最好的工具

这个例子说明了我以前曾经提过的: 没有一个工具是万能的、能做好任何事。这个拍卖应用如果只是使用 JSP 脚本, 那将很难实现。只使用 applet 将根本不能工作 (除非它们已经签过名, 即使是那样也很难实现)。一个定制的客户机/服务器系统可能有效, 但是那将更难于创建这样一种系统。

但是, 结合上述三种技术: 一个 Java 服务器、applet 以及 JSP, 会产生一个很好的系统, 并且不难于开发。不要盲从于一种工具或者是技术, 它可能常常让你的生活更加不顺, 甚至限制你的工作能力。

13.2 快速解决方案

13.2.1 使用 JSP 进行服务器端编程

当你需要处理 Web 浏览器请求时, 你的第一反应可能是写一个定制的服务器。尽管这是一种选择, 使用一个支持 JSP 的商用服务器常常更有效。

JSP 服务器允许你使用称作 servlet 的小 Java 程序对 Web 服务器进行扩展。有些专门的 Java 程序可以与用户的浏览器进行交互。但是, 因为 servlet 写起来有点复杂, 你可使用 JSP 来代替。JSP 能让你在 HTML 文件里嵌入 Java 代码。服务器将 JSP 编译成一个 servlet, 接着将 servlet 编译成普通的 Java 类文件。接着服务器可以执行 servlet 并处理用户请求。

有些 JSP 服务器是独立的, 其他一些服务器要附加在另外的 Web 服务器上才能运行。对开发人员来说, 有几种比较流行的选择方案:

- Tomcat——是 Apache 工程中的 JSP 服务器。
- Jrun——来自 Allaire 的 JSP 服务器, 可以与大多数流行的服务器集成。

13.2.2 从 JSP 中读输出

像任何网页一样, JSP 可以从浏览器中接收数据, 或者是查询串, 或者是一个 HTTP 的 POST 方法的表单数据。查询串是出现在 URL 里跟在问号后边的一个或者多个可变参数。例如, 请看这个 URL:

```
http://www.coriolis.com/test.jsp?fname=Nick&lastname=Rivers
```

这个例子中的整个查询串是 `fname=Nick&lastname=Rivers`。有两个查询串参数: `fname` 和 `lastname`。记住你必须对 URL 中有特殊意义的字符进行编码。例如, 如果你想设置名为 `fullname` 的变量, 值为 `Nick Rivers`, 你必须将它编码成:

```
Fullname=Nick+Rivers
```

静态函数 `URLCoder.encode` 可以执行这种类型的编码。

浏览器可以传递信息给网页的另一种方法是使用表单和 `POST` 方法。如果你在表单中使用 `GET` 方法，数据会包装在请求的查询串中。

当使用 `POST` 时，数据已经编过码，位于 `HTTP` 请求的内容部分。当使用 `JSP` 时，你可以轻易地读取查询串或者一个 `post` 的参数。你使用的 `JSP` 调用要将数据分成几个独立的变量，并对所有特殊的编码进行解码。

下面是一小片 `JSP` 代码，读取 `fname` 参数。`JSP` 查找表单数据和查询串。如果它不能定位名为 `fname` 的参数，它将返回一个 `null`。

```
String userName = request.getParameter("fname");
```

13.2.3 在 JSP 中向浏览器写数据

当你想回送数据到浏览器时，有两种选择：

- 可以使用 `out` 对象。`out` 对象引用 `PrintWriter` 的一个特殊类型，允许你向 `HTML` 输出流里写数据。
- 使用特殊的标签对。你可以使用 `<%=` 和 `%>` 标签来封装一个单独的表达式，它可以将数据传递给浏览器。

应该使用哪种形式呢？它依赖于一定条件。如果是处在代码中比较复杂的部分，并需要产生输出，将很可能要使用 `out` 对象。例如，看看一个 `JSP` 中的这段代码：

```
<HR>
<%
    int x=func1();
    int y=func2();
    out.println("Result=" + x + " and " + y + "<BR>");
    int z=func3();
    out.println("Z=<B>" + z + "</B><BR>");
%>
```

另一方面，如果页面的主要流程集中在 `HTML` 上，那最好使用标签对的形式。例如：

```
<% Date today = new Date(); %>
Hello, today is <%= today.toString() %>
```

13.2.4 使用 JSP 页的定向功能

另一个特殊的标签是 `<%@` 标签。这对于引入 `import` 语句很有用。例如：

```
<%@ page import="java.io.*" %>
<%@ page import="java.util.*" %>
```

13.2.5 写一个简单的 Web 服务器程序

一个 `Web` 服务器不会比其他章中的服务器更复杂。`Web` 服务器必须执行三个基本步骤：

1. 接收请求（典型的在 80 端口上）。

2. 发送响应标题给请求者。
3. 发送请求的文档给请求者。

对一个简单的 HTTP 请求，甚至不用发送标题。

清单 13.3 中的代码是 `HttpServer` 类。它执行上述步骤，你可能使用它作为一个基类来定制服务器。

服务器只接收 GET 请求，它假设每个请求包含两个或者三个部分：

- 第一部分——单词 GET。
- 第二部分——浏览器请求的文档。
- 第三部分（可选）——HTTP 版本号。

服务器实际上不对第三个参数进行解释。如果有第三个参数，服务器假设 HTTP 请求版本为 1.0 或者更高。在这种情况下，服务器以 1.0 来响应（它支持的最高版）。如果没有版本标志，服务器假设请求是一个简单的 HTTP 请求（也称作是 HTTP0.9 请求）。在这里，服务器不接受标题，也不产生标题。

13.2.6 配置 HttpServer (Http 服务器)

`HttpServer` 类可以充当你自己 Web 服务器类的基础。服务器依赖于两个属性文件来控制它的操作。第一个文件是 `server.properties`，可以包含三个参数：

- `portnumber`——服务器监听请求的端口。
- `root`——文档的根目录。
- `defaultdocument`——用于浏览器请求一个目录时的文档名。

第二个文件是 `mimetypes.properties`。它包含显示 Content-Type 标题的关键字，与文件的扩展名相对应。例如：

```
htm=text/html
html=text/html
jpg=image/jpeg
jpeg=image/jpeg
gif=image/gif
```

13.2.7 定制 HttpServer

你可以写 `HttpServer` 的子类以提供客户化的行为。为简化重用，对象提供了 `action` 例程。一个子类可以重载这个方法提供客户化处理。

当服务器识别出一个合法的请求时，它调用 `action`，传给它两个参数：请求中的文件名以及一个 `Hashtable`，包含传给请求的任意请求串。

你的 `action` 方法返回一个文件名，服务器会将这个文件名返回给浏览器。最简单的 `action` 方法将传给它的文件名参数返回。但是，这个方法也可以识别特殊的文件名，并执行任何类型的请求处理。接着，如果必需，你可以改变服务器将提供的文件名。

例如，请看清单 13.12。这个程序识别特殊的文档名 `time`。action 例程将当前时间和日期写进 `time.htm` 文件，并将它用作请求文件。下面是实现该工作的代码：

```
if (fn.equals("/time")) {  
    PrintWriter tout=  
        new PrintWriter(  
            new FileWriter(props.getProperty("root")+"/time.htm"));  
    Date d=new Date();  
    tout.println(d);  
    tout.close();  
    return "/time.htm";  
}
```

当然，如果很多用户立即请求时间，你可能在有人读 `time.htm` 的同时要对该文件进行写操作，这样会产生问题。

清单 13.12 该服务器使用定制的 action 方法来处理时间请求

```
// Time Server  
// Each request to the special file  
// "time" will cause time.htm to  
// get the current time and  
// then serve that file name  
// probably not safe for multiple accesses!  
import java.util.*;  
import java.io.*;  
  
public class TimeHttp extends HttpServer {  
    public String action(String fn, Hashtable vars) {  
        try {  
            if (fn.equals("/time")) {  
                PrintWriter tout=  
                    new PrintWriter(  
                        new FileWriter(props.getProperty("root")+"/time.htm"));  
                Date d=new Date();  
                tout.println(d);  
                tout.close();  
                return "/time.htm";  
            }  
        }  
        catch (Exception e)  
        {  
            System.out.println("Except: " + e);  
            return "err.htm";  
        }  
        return fn;  
    }  
  
    public static void main(String[] args) {  
        new TimeHttp();  
    }  
}
```

提供时间的一个更好的方法是为服务器的发送提供一个串 (String)。即使那样, 你仍然需要将 `time` 关键字转换成一个 .htm 文件, 这样服务器可以正确地猜出 MIME 类型。但是, 这样, 服务器实际上不会打开该文件。

写这种类型服务器的关键是使用 `openFile` 方法, 为服务器提供一个要读的串。清单 13.13 显示了一个简单的实现。这个程序还提供了一个 `action` 方法, 如果方法要提供客户化处理, 那就要将处理标志设为 `true`。

实际工作发生在 `openFile` 中。当处理标志为 `true` 时, 代码创建一个带有当前时间和日期的串, 并返回一个映射该串内容的 `DataInputStream`。编译器将会提示 `StringBufferInputStream` 是反对使用的, 但是因为原始的服务器代码使用了 `DataInputStream`, 你可以在原始的代码中使用更新的 `StringReader` 对象, 不需要进行重大的改动。

如果 `process` 变量为 `false`, 程序只调用基类的方法。这允许像以前一样的普通的处理请求。

清单 13.13 该服务器使用一个 String 来提供当前时间

```
// This server subclass uses a string to send the browser
// dynamic content
import java.util.*;
import java.io.*;

public class Time1Http extends HttpServer {
    private boolean process=false; // special request?
    public String action(String fn, Hashtable vars) {
        if (fn.equals("/time")) {
            process=true;
            return "/time.htm"; // fake name
        }
        process=false;
        return fn;
    }

    public DataInputStream openFile(String fn, Hashtable vars)
        throws IOException {
        if (process) {
            String res = new Date().toString();
            // StringBufferInputStream is deprecated, but can't use StringReader
            // without changing base class
            return new DataInputStream(new StringBufferInputStream(res));
        }
        else
            // do original code
            return super.openFile(fn, vars);
    }

    public static void main(String[] args) {
```



```

        new Time1Http();
    }
}

```

13.2.8 写一个代理服务器

最简单的代理服务器的形式是同时充当一个 Web 服务器和一个 Web 客户端。当用户让浏览器使用代理时，它发送所有的 HTTP 请求给代理，而不是发送请求给 Web 服务器的 80 端口。代理的工作是将请求发送到正确的服务器（除非它不接受该请求）。接着，代理也必须将从服务器返回的数据传递给客户端。

下述任务是比较有用的：

- 对 Web 服务器的内容进行缓存。
- 不允许访问一定的网站。
- 对来自某网站的内容进行过滤。
- 对网页内容进行翻译或者格式化。
- 对 Web 的访问进行登录。

清单 13.5 显示了一个代理服务器的基类，你可以直接使用该类，或者你也可以继承一个新类来对服务器的操作进行客户化。表 13.1 总结了可获取的方法。

表 13.1 代理的基类提供的一些有用的方法和变量

变量/方法	说明
BUFSIZ	套接字输入缓冲的大小
CONNECT_PAUSE	两次连接之间的暂停时间
CONNECT_RETRIES	在放弃连接之前试图连接远程主机的次数
Log	一个 OutputStream 对象，用于缺省日志例程，记录登录信息
Logging	是否需要代理来记录所有传输的数据（如果为 true 则表示要记录）
SetParentProxy	允许将本代理链向另一个代理（要指定另一个代理的名字和端口号）
TIME-OUT	等候套接字输入的时间

为创建一个定制版本的服务器，采取如下步骤：

1. 创建 HttpProxy 的一个子类。
2. 提供对 writeLog 的一个重载，如果你愿意记录字符的话。该方法接收两个参数：c 是正发送的字符，如果字符来自浏览器，那么浏览器标志（一个布尔变量）为 true。
3. 提供对 processHostName 方法的重载。该方法接收关于 HTTP 请求的数据，必须返回满足请求的主机名。该方法可以改变主机（尽管这不常见），并且，它也可以对浏览器请求的

URL 进行日志记录。

4. 写一个合适的 main 例程，使用代理的端口号和继承的类对象来调用 startProxy 方法。该类对象与定制的代理的代码相对应。

5. 提供一个构造函数，能恰当地对基类进行初始化。通常，它使用一个 Socket 作为参数，通过 super 简单地将它传给基类的构造函数。

你可以在清单 13.14 中找到简单的定制代理的例子。这个代理不为主机做任何事情，但是它确实将它处理的字符写出来了。

清单 13.14 该代理写一个简单的日志文件

```
// Trivial subclass of HttpProxy
// Does not log hostnames and
// puts a * in front of each line of log output

import java.io.*;
import java.net.*;

public class SubHttpProxy extends HttpProxy {
    static private boolean first=true;
    public SubHttpProxy(Socket s) {
        super(s);
    }
    public void writeLog(int c, boolean browser) throws IOException {
        if (first) log.write('* ');
        first=false;
        log.write(c);
        if (c=='\n') log.write('* ');
    }
    public String processHostName(String url, String host, int port, Socket sock)
{
    // do nothing
    return host;
}
// Very simple test main
static public void main(String args[]) {
    System.out.println("Starting Subproxy on port 808<BR>");
    HttpProxy.log=System.out;
    HttpProxy.logging=true;
    HttpProxy.startProxy(808,SubHttpProxy.class);
}
}
```

13.2.9 调试一个代理服务器

代理服务器的另外一种用法是用于调试 Web 应用程序。你可以创建一个简单的代理，将

所有通信从控制台输出或者记录到文件当中。这可以让你看到数据从服务器流入流出的完整传输脚本。那些数据包含 cookie 和其他标题。

作为一个例子，想象你正为一个使用 cookie 的程序犯难。要准确地看看服务器设置的是哪一个 cookie 以及浏览器接收的是哪一个 cookie 常常很困难。有关 cookie 的内容详见第 10 章。

你可以写一个代理服务器，当服务器发送 cookie (Set-Cookie 标题) 的时候，或者是客户端返回 cookie (Cookie 标题) 给服务器的时候，就将它们打印出来。代理服务器出现在清单 13.15 中。它创建两行 (在 line 数组中)：一行用于到浏览器的数据，另一行用于来自浏览器的数据。当服务器发现行结束符时，它检查行，看看它是否可能是 cookie 相关的标题。

清单 13.15 该代理服务器用于显示 cookie 事务

```
// Show cookie headers
import java.io.*;
import java.net.*;

public class CookieProxy extends HttpProxy {
    static private StringBuffer line[];
    public CookieProxy(Socket s) {
        super(s);
        line=new StringBuffer[2];
        line[0]=line[1]=null;
    }
    public void writeLog(int c, boolean browser) throws IOException {
        int index=browser?1:0;
        if (line[index]==null) line[index]=new StringBuffer();
        if (c=='\r' || c=='\n') {
            // Heuristic guesses when it sees a cookie header
            // not perfect, but good enough
            if (line[index]==null) return; // nothing
            if (line[index].length()<(browser?7:11)) return;

            if (line[index].substring(0,browser?6:10).
                compareToIgnoreCase(browser?"cookie":"set-cookie")==0) {
                System.out.print(browser?"Sent ":"Received ");
                System.out.println(line[index]);
            }
            line[index]=null;
            return;
        }
        line[index].append((char)c);
    }
    public String processHostName(String url, String host, int port,
        Socket sock) {
        // do nothing
    }
}
```

```
        return host;
    }
    // Very simple test main
    static public void main(String args[]) {
        System.out.println("Starting Cookieproxy on port 808<BR>");
        HttpProxy.log=System.out;
        HttpProxy.logging=true;
        HttpProxy.startProxy(808, CookieProxy.class);
    }
}
```

因为服务器只是用于调试，我决定不再重点考虑设计，服务器使用启发式识别 cookie 标题，可能会显示一些错误的观点（例如，如果单词 Cookie 出现在文档中的行的开头）。在实际中，这不会是问题，因为你很少产生错误的点击，如果确实得到这样的结果，那也是很明显的。

注意：像这样的代理服务器效率不太高，因为它为每个字符创建了行。为了调试，这不是问题。但是，你不会让代理服务器在所有时间里都有效，除非你喜欢让网上冲浪变缓慢。同时，你产生越多的调试输出，代理的操作就会越慢。

当然，代理直到你做了下面几步以后才开始工作：

1. 在运行任何 Java 程序时，运行代理服务器。
2. 设置浏览器，使用代理服务器，并为它指定服务器使用的端口号。对于最新版本的 IE，你可以使用“工具\Internet 选项”菜单，再选择连接属性页，在那里可以输入代理的 IP 地址（127.0.0.1，如果你只使用一台机器）以及端口号（缺省为 80）。

代理一运行，你将看到 Java 控制台的输出，显示了发送的 cookie。例如，下面是来自美国的 Today 网站的一个输出：

```
Sent Cookie: RMID=cfdaf5063a37e340
```

大多数网站不再会设置 cookie，除非不得已。这样你可能必须要改变客户化的页或者对站点作其余的改变，看看站点发送给你的 cookie。例如，当我改变我的参数选择时，www.alltheweb.com 站点送下列 cookie：

```
Received Set-Cookie: PREF=cs=iso-8859-1:cw=lang:l=any:no=off;
    expires=Tue, 30 Jul 02 14:29:58 GMT; path=/
```

第 14 章 XML

14.1 深入介绍

你可以通过看科幻小说学到很多知识。当我还是小孩子的时候，就看了很多这种小说。我最喜欢的作者之一就是 Harry Harrison。他的书有些非常幽默（像 Stainless Steel Rat 系列）。他的 Eden 故事集就显得严肃得多了。而 Deathworld trilogy 故事就介于上述两者之间。

Harrison 的故事特征比较突出的就是 Esperanto（世界语）了。世界语是由 Ludwig L. Zamenhof 医生（回忆一下，是一位眼科医生）发明的一种语言。这位医生感觉世界需要一种通用语言，它很容易学、容易拼，并且容易发音。结果产生了世界语。有一段时间，看起来世界语可能会作为一个通用的第二语言而流行起来。现在你很少听到世界语了，但是世界上仍有一小部分人讲世界语（估计它与希伯米语或者立陶宛语一样普通）。还有世界语杂志、报纸甚至是世界语会议。

通用语言刚好可以与 Web 和 Internet 作比较。Internet 使得全世界的人都很容易联系。但是，它实际上并不强调人们的互相理解。例如，HTTP 确保你可以坐在你的起居室里，从 Burkina Faso（这个你应该知道，是有关非洲的 Colorado 面积的网站）上下载一个网页。但是，如果那个页面使用的是苏丹语方言（那个网站有可能使用这种语言），你就可能读不懂它。

尽管可以使用机器翻译（例如，<http://babelfish.altavista.com>），它们的功能很不完善。对于人类语言而言，这可能是在最近的将来能想象得到的最好结果——对重要语言的不完善的翻译（我怀疑 Babel Fish 是否能翻译苏丹语）。

但对于机器而言，就不会有这么多困难。为何一台使用 DB2 的 IBM 大型机与使用 Access 的 PC 机共享数据就那么困难呢？要解决这个问题，很多应用都求助于扩展的标识语言（XML）。从表面上看，XML 看起来像是 HTML，但是它们在语法上有些微妙的不同。

然而，XML 与 HTML 较大的不同在于它们解决的不是相同的问题。很多人误以为它们是紧密相关的，因为 HTML 看起来与 XML 相似。事实上，HTML 与 XML 是完全不同的两码事。

当然，HTML 可以让你对你喜爱的页面进行建模，并显示或者打印，通常来自一个 Web 浏览器中。从人类的消费角度来讲，这已经工作得很好，因为 HTML 允许你对标题、链接以及嵌入的图像进行定义。

看看下面这个网页：

```
<html>
```



```

<body>
<h1>Java Network Programming Black Book</h1>
<P>We have 10 of these paperback books by Al Williams.</P>
<h2>Shelving</h2>
<ul>
<li>Computer
<li>Java
<li>Programming
</ul>
</body>
</html>

```

至少对人而言，这个网页的意思很明显。你可以很容易地判断作者的姓名，还有供应商的存货里有 10 本书，并且书是用纸包装的。并且，你可以看出供应商认为书是放在架子上的。

从机器的角度来讲，这个页面就没有那么理想了。例如，`<H1>` 标签，很明显地引入了书的标题，但是，它代表不了什么内容。正文中的数据是自由表单，虽然你可以通过应用某些启发式规则来辨别含义，页面内容的任何变化将会混淆大多数程序。

问题是标签不会调查内容的含义，它们只是标识文档的部分。浏览器想知道有关标题的情况，但是解释数据的程序想知道的是内容，而不是显示。

使用 HTML 的另一个问题是页面的内容和显示是混合在一起的。假设你有一系列网页，里边有很多本书，一个文件与一本书相对应。如果你想产生那些书的列表，那该怎么办？或者你可能想从文件中剥离信息，使用文本语音转换技术来发电话信息时，又怎么办？

14.1.1 进入 XML

XML 试图通过集中表示数据来解决这些问题。如果你想显示数据，需要告诉浏览器如何对数据进行格式化。格式化对于 XML 来说并不重要。

XML 确实使用了与 HTML 类似的标签结构。但是，它实际上没有预定义的标签。你（以及数据的接收者）定义自己的标签。清单 14.1 显示了一个简单的 XML 文档。

清单 14.1 这个简单的 XML 文件描述了一本书

```

<?xml version='1.0' encoding='UTF-8'?>
<book title="Java Network Programming Black Book" author="Al Williams">
  <quantity>10</quantity>
  <paperback/>
  <shelving>
    <shelve>Computer</shelve>
    <shelve>Java</shelve>
    <shelve>Programming</shelve>
  </shelving>
</book>

```

这个文档包含了 HTML 例子中的完全相同的信息。但是，它的格式是明确的。假设程序读到这个文件，意识到使用的标签，它就可以精选有关这本书的数据。

如果你试图将 XML 文件看作是一个能理解 XML 的 Web 浏览器，你将看到浏览器使用彩色编码的元素显示源代码，并标出层次，不是用户想要看到的那种视图。使用特殊类型的样式单可以将 XML 文件显示成不同的方式。但是大多数人都使用一种特殊的格式语言，即所谓的扩展样式语言（XSL）来完成这个任务。

14.1.2 XML 语法

清单 14.1 看起来几乎像是使用手工标签的 HTML 文件。但是，关于语法（大概一看可能很不明显）有若干关键点要指出：

- 第一行是一个<?xml>标签。这个标签将文件标识成一个 XML 文档，并标识了文档使用的 XML 的版本号。
- 第一个标签是文档的根标签。这与<HTML>标签类似，因为它包含所有其他标签。在 XML 里，这个标签不是可选的，尽管你可以将它命名成你想要的名字（在清单 14.1 中它是一个<book>标签）。
- 所有标签都有一个结束标签。如果在标签的开始和结束之间没有内容，你可以省略它。例如，<paperback/>标签既有开始又有结束，它与<paperback></paperback>等价。
- 标签区分大小写。
- XML 标签的嵌套必须恰当。如果文件包含一个标签<A>，后跟一个标签，那么必须出现在标签之前。
- XML 文档保留空格。这个规则的例外是回车符和行符成为单行符。
- 属性必须以单引号或者双引号引起来。即使它们不包含空格，你也要这样做。

如果你习惯了手动编写 HTML，那么引号很难记住，在 HTML 里下面的写法是合法的：

```
<IMG SRC=mylogo.gif>
```

如果其中有空格，你就必须使用引号：

```
<IMG SRC="my new logo.gif">
```

使用 XML，你必须总是使用引号，即使值没有空格或者其他特殊字符。

因为这些严格规则，程序可以很容易地解析 XML 文件，即使它不理解使用的标签。写一个 HTML 解析器要难得多，因为大多数浏览器将接收嵌套不标准的标签，甚至接受那些本应该有结束标签的标签。在这些地方，就很难决定如何处理。例如：

```
<P><B><I>Hello</B><P>There</I>
```

甚至是使用<P>、和<I>标签的一个 XML 文档，那部分的代码也是非法的。当然，在 HTML 里，它在技术上也是非法的。但是大多数浏览器都能解释。XML 程序就不会这么宽松。你不得不将它改成下列形式（忽略根标签）：

```
<P><B><I>Hello</I></B></P><P><I>There</I></P>
```

虽然你可以造名字，还是有些规则要遵守：

- 名字不能以数据或者标点符号开头。
- 名字不能包含空格，并且不能以 XML 开头（这是不区分大小写的，因此名字不能

以 xml、XmL 或者任何其他的 XML 拼写组合)。

- 同时要避免在名字中使用冒号。

提示: 虽然 XML 与 HTML 不同, 将 HTML 标签定义成 XML 版的对应标签。要做到这样, 你必须减少没有结束标签的标签 (如
), 并且需要引用以及对标签的嵌套进行校正。结果产生了有关 HTML 的 XML 友好版本, 那就是 XHTML。XHTML 要求标签名字都是小写, 因为 XML 区分大小写。

14.1.3 有效的 XML

如果你能够编写自己的标签, 那你又如何知道是否是创建有效的 XML 文件呢? 很明显, 如果你是试图读数据的程序, 只要看看是否缺省需要的标签或者是否有你不能理解的标签。但是, 只需要验证 XML 文件是否有效的程序该如何写呢? 例如, 一个知道如何导入 XML 的编辑器需要知道文件是否是正确有效, 但并不真正在意文件中的内容。

XML 文档的结构有多种定义方式。一种方式是 DTD (或者称为文档类型定义)。这是另一个文档, 用于定义任意 XML 文档的结构。

清单 14.2 使用嵌入的 DTD 显示了相同的关于书的 XML 文档。你也可以引用单独文件里的 DTD, 见清单 14.3。这是一个相同的 XML 文件, 但是没有使用内部的 DTD, 它使用了清单 14.4 中的 DTD。

清单 14.2 使用嵌入式 DTD 的 XML 文档

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE book [
  <!ELEMENT book (quantity,(paperback|hardback),shelving)>
  <!ATTLIST book title CDATA #REQUIRED >
  <!ATTLIST book author CDATA #REQUIRED >
  <!ELEMENT quantity (#PCDATA)>
  <!ELEMENT shelving (shelve+)>
  <!ELEMENT shelve (#PCDATA)>
  <!ELEMENT paperback EMPTY>
  <!ELEMENT hardback EMPTY>
]>

<book title="Java Network Programming Black Book" author="Al Williams">
  <quantity>10</quantity>
  <paperback/>
  <shelving>
    <shelve>Computer</shelve>
    <shelve>Java</shelve>
    <shelve>Programming</shelve>
  </shelving>
</book>
```

!DOCTYPE 标签标识了嵌入定义。每个 !ELEMENT 标签定义了单个 XML 标签, 并包含

了在这个标签内有可能出现的内容信息。特殊标志`#PCDATA` 表明是正常文本，XML 将为标签和特殊字符作解析（与`#CDATA` 相反，它不用解析）。注意 `paperback` 和 `hardback` 标签使用了 `EMPTY`，因为它们不包含任何数据。

`book` 的 `!ELEMENT` 标签定义了一种选择。其中可能有一个 `paperback` 标签或者一个 `hardback` 标签，但不能同时存在。类似，`shelving` 标签可以有一个或者多个 `shelve` 标签（由加号指定）。不使用加号，你也可以使用星号表示 0 个或者更多，或者用一个问号表示 0 个或者 1 个。

清单 14.3 使用外部 DTD 的一个 XML 文档

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE book SYSTEM "book.dtd">
<book title="Java Network Programming Black Book" author="Al Williams">
  <quantity>10</quantity>
  <paperback/>
  <shelving>
    <shelve>Computer</shelve>
    <shelve>Java</shelve>
    <shelve>Programming</shelve>
  </shelving>
```

清单 14.4 该文件是一个外部 DTD

```
<!ELEMENT book (quantity,(paperback|hardback),shelving)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT shelving (shelve+)>
<!ELEMENT shelve (#PCDATA)>
<!ELEMENT paperback EMPTY>
<!ELEMENT hardback EMPTY>
<!ATTLIST book title CDATA #REQUIRED >
<!ATTLIST book author CDATA #REQUIRED >
```

除了 `!ELEMENT` 标签以外，DTD 也定义了标签是否可能拥有属性。在清单 14.4 中，有两个属性都属于 `book` 标签。两个属性 `title` 和 `author`，包含字符数据，都没有缺省值。它们都是必需的，这由 `#REQUIRED` 关键字来指定。

有一件事要注意的是 DTD 文档不是 XML。使用另一个 XML 文件来描述一个 XML 文件是可能的。这就是所谓的 XSchema 或者有时称为 Schema（纲要）。一个 XSchema 文件与 DTD 可以用作相同的服务，但是它是用 XML 文件编写的。那意味着你可以使用相同的工具和技术来编写 XSchema 文件和其他任何 XML 文件。通常情况下，这些纲要文件的扩展名为 `.xsd`。

你可以读关于 XML 的整本书。但是，这些基础将允许你从 XML 和 Java 起步。在你的 Java 程序与 XML 结合时，有两个任务要做。一个是创建 XML 文件，另一个是读（解释）XML 文件。

14.1.4 文档对象模型 (DOM)

将 XML 文件看作是文本也许不太明显，但是 XML 数据很自然地组织成树状结构。每个 XML 文件都有一个根标签，这个标签可以是树结构的根，根标签带有预定义的子标签，那些子标签可能还有自己的子标签。

表达这种思想的一种常见方式是 DOM (文档对象模型)。因此清单 14.3 中的例子中有 7 个结点：book、quantity、paperback、shelving 以及三个 shelf 结点。book 结点是根结点，并有三个子结点 (quantity、paperback 和 shelving)。shelving 结点有三个 shelf 结点作为它的子结点。

很明显，可以将这些结点用作 Java 对象。接下来，你将看到专门的 Java 库是如何从一个 XML 文件中创建一个 DOM 的。相反，你也可以创建一个 DOM，并使用它正确地输出一个 XML 文件。

14.1.5 名字空间

关于 XML 的另一个重要思想是名字空间。因为你可能定制自己的标签名，那么混合两个或者更多的 XML 文档可能会出现冲突。假设你要创建一个文档，它包含来自两个不同的 XML 的 DTD (文档类型定义) 中的元素。这种情况是很常见的，例如，当创建样式单或者创建复杂的组合文档的时候就会产生这种情况。

考虑你有一个定义了名为 address 的标签的 DTD。address 标签可能在一个像这样的文件里出现：

```
<address>
<street>14455 North Hayden Road</street>
<suite>220</suite>
<city>Scottsdale</city>
<state>AZ</state>
<zip>85260</zip>
</address>
```

但是，假设另有一个 DTD 定义了一个 IP 地址，它也使用了 address 标签：

```
<address>
<ip>38.187.128.45</ip>
<host>coriolis.com</host>
</address>
```

如果你混合这些标签，就不会有问题。但假设你想将两者的元素合并到一个文件中（假设你想将 Web 地址与物理地址联系起来）。不经过预先扫描，程序如何知道你要使用的是哪一个标签呢？

答案是使用名字空间。从技术上看，名字空间是一个看起来像 URL 的串，URI (统一资源标识) 唯一地标识了一个标签集合。URI 并不真正指向一个 Web 资源——你可能将 `http://x-y-z` 作为一个 URL——它只是一个唯一的标识符。因为 URI 一定要全球唯一，你应该

使用一个你拥有的域名来避免与其他人的 URL 相冲突。

看看这篇文档：

```
<address xmlns="http://www.al-williams.com/xml/address">
<street>14455 North Hayden Road</street>
<suite>220</suite>
<city>Scottsdale</city>
<state>AZ</state>
<zip>85260</zip>
</address>
```

这个 URL (xmlns 属性的一部分) 唯一地标识了这个标签的集合。接着你可能像下面这样写一个 IP 地址：

```
<address xmlns="http://www.coriolis.com/xmlns/ipaddress">
<ip>38.187.128.45</ip>
<host>coriolis.com</host>
</address>
```

现在很明显，各个标签都指的是哪个。在这里，由 xmlns 属性表示的名字空间对 address 标签包含的所有标签都适用。将一个本地名附给一个名字空间也是可能的，如下所示：

```
<staddress:address xmlns:staddress="http://www.al-williams.com/xml/address">
<staddress:street>14455 North Hayden Road</staddress:street>
<staddress:suite>220</staddress:suite>
<staddress:city>Scottsdale</staddress:city>
<staddress:state>AZ</staddress:state>
<staddress:zip>85260</staddress:zip>
</staddress:address>
```

当你将两项合并到一篇文档里时，这显得最有用：

```
<webhost xmlns:staddress="http://www.al-williams.com/xml/address"
          xmlns:ipaddress="http://www.coriolis.com/xmlns/ipaddress">
<staddress:address> . . . </staddress:address>
<ipaddress:address> . . . </ipaddress:address>
```

即使你不打算将 XML 文档混合在一起，在使用其他基于 XML 的文档如 JSP 等，你也可能遇到名字空间的问题。

14.1.6 Java 对 XML 的支持

在 Java 程序或者 JSP 脚本中使用 XML 有多种方式。当然，你可以只用手创建 XML，但是那样就利用不了多少 Java 的特性，同时也不能确保正确的文档形式。但是，作为一个起点，看看清单 14.5 中的 JSP 脚本。这与以前的 XML 文档相同，除这次以外，大多数关键值都是 Java 值。在这个例子里，我只是将变量值设为常数值。在实际生活中，你还要将变量从数据库中取出或者另外动态生成变量。

清单 14.5 一个使用 JSP 的动态 XML 文档

```
<?xml version="1.0" ?>
<%@ page contentType="text/xml" %>
```

```

<%
// fake database read
String title="Java Network Programming Black Book";
String author="Al Williams";
int quan=33;
boolean paper=true;
%>

<!DOCTYPE book [
  <!ELEMENT book (quantity,(paperback|hardback),shelving)>
  <!ELEMENT quantity (#PCDATA)>
  <!ELEMENT shelving (shelve+)>
  <!ELEMENT shelve (#PCDATA)>
  <!ELEMENT paperback EMPTY>
  <!ELEMENT hardback EMPTY>
  <!ATTLIST book title CDATA #REQUIRED >
  <!ATTLIST book author CDATA #REQUIRED >
]>

<book title="<%= title %>" author="<%= author %>">
  <quantity><%= quan %></quantity>
  <% if (paper) { %>
    <paperback/>
  <% } else { %>
    <hardback/>
  <% } %>
  <shelving>
    <shelve>Computer</shelve>
    <shelve>Java</shelve>
    <shelve>Programming</shelve>
  </shelving>
</book>

```

即使这是一个 XML 文件，它的扩展名为.jsp，这样服务器将它当作 JSP 来处理，能意识到这一点是很重要的。page 定向指定的 text/xml 内容类型是很重要的。没有这个语句，JSP 就不会产生一个 HTML 文件。

这个文档中的四个变量分别是 title、author、quan 和 paper。除了 paper 外，它们都只出现在脚本里。paper 变量是 boolean 型，它控制着文档是否包含一个<paperback/>标签或者一个<hardback/>标签。

创建一个 JSP 文件，再输出 XML 文件，是创建 XML 文档的一种方式。你可以在现在大多数浏览器中浏览 XML 文件，但是要想看到格式化的 XML，还得需要样式单或者 XSL 文档将 XML 转换成网页。尽管这很简单，Java 有更多非常有用的方法来处理 XML。

14.1.7 XML 库

你可以使用多种 Java 技术来处理 XML:

- JAXP——用来解析 XML 的 Java API, 允许你读 XML 文件, 并可以用不同的方式对它们进行解释。
- JAXB——用于 XML 绑定的 Java 体系结构, 允许你将一个 Java 对象写成一个 XML 文件, 并在以后使用相同的 XML 文件进行重构。
- JDOM——用于将 XML 文件转换成一个树状的 Java 对象的 Java 文档对象模型。
- JAXM——用于 XML 消息的 Java API, 允许程序交换基于 XML 的消息。
- JAXR——用于 XML 注册的 Java API, 允许你为其他寻找 XML 服务的程序发布存在的 XML 服务。

Sun 公司知道很多开发人员需要下载所有的 XML API 函数及相应的库。这就是为何创建 JAX 包 (<http://java.sun.com/xml/jaxpack.html>) 的原因。从该页上, 你下载一次就可以得到所有的相应库。

JAXP 允许你使用简单的 API 来做 XML 解析 (SAX 解析器) 或者你也可以使用 `DocumentBuilder` 对象从 XML 输入中创建一个 DOM。有一个重要的地方要注意: JAXP 是 XML 解析器的一个接口, 它实际上不是 XML 解析器 (虽然有多个解析器与 JAXP 绑在一起)。这个很重要的原因是你可以自由地替换其他解析器。只要它们使用 JAXP 工作, 你的程序不用改变。你可以通过设置系统属性来控制使用的解析器。 `javax.xml.parsers.SAXParserFactory` 属性控制解析器的使用, 而 `javax.xml.parsers.DocumentBuilderFactory` 属性用于处理 `DocumentBuilder` 对象。

有多个库组成了 JAXP 的关键部分:

- `javax.xml.parsers`——主 JAXP 接口。
- `org.w3c.dom`——文档对象模型类。
- `org.xml.sax`——SAX 接口。
- `javax.xml.transform`——允许你将 XML 转换成其他格式。

SAX 接口与 DOM 接口之间有何区别呢? 像我以前提到的一样, DOM 接口只是为你展示一个完整的对象树, 其中一个对象对应一个标签, 这个方法的问题在于做任何处理之前, 你不得不读整个 XML 文档。

另一方面, SAX 接口通过逐个元素来处理 XML 文件。当 SAX 解析器发现不同类型的元素时, 它会调用你提供的特殊的方法来通知你。你可以通过 SAX 立即开始处理一个 XML 文件, 即使你还没有完全的 XML 文档。

JAXP 需要你使用若干个 JAR 文件。表 14.1 显示了你需要的文件以及它们包含的内容。

表 14.1 JAXP 使用三个不同的 JAR 文件以及多个不同的包

Jar 文件	包	内容
jaxp.jar	javax.xml.parsers, javax.xml.transform	Interfaces
crimson.jar	org.xml.sax, org.w3c.dom	Interfaces and helper classes
xalan.jar	javax.xml.parsers, javax.xml.transform, org.xml.sax, org.w3c.dom	Implementation classes

14.1.8 使用 SAX

如果你想读一个 XML 文件，并检查所有的不同部分，你将可能需要一个基于 SAX 的解析器，例如清单 14.6 中的解析器。程序使用 import 来包含要使用的重要的 XML 库。另外，你还需要确信你的 CLASSPATH 包含 jaxp.jar 和 crimson.jar 来运行这个程序。

XMLEcho 类扩展了 DefaultHandler——一个帮助类，它有占位符，供解析器将要调用的方法使用。每次解析器发现 XML 输入的一个元素，它调用回调对象（在这里是 XMLEcho）相应的方法。

首先，你需要得到 SAX 解析工厂的一个实例，通过调用 SAXParserFactory.newInstance 可以得到。使用这个对象，你可以调用 newSAXParser 来创建一个真实的解析器对象。接着，你调用 parse 来启动解析的处理过程。parse 方法需要一个 File 对象以及对一个回调对象的引用。

表 14.2 显示了 SAX 解析器可能调用的回调方法。因为 DefaultHandler 提供了对这些方法的实现，XMLEcho 只提供了它需要使用的那些方法。

清单 14.6 中的例程在标准输出（使用 emit 方法）上重新创建了对 XML 文件的拷贝。它也报告了关于标准错误流里有关文件的信息。这样你可以按如下所示来运行程序：

```
java XMLEcho test.xml >testout.xml
```

这将可以让你在屏幕上看到输出的报告，同时将 XML 文件保存到另一个 XML 文件。

除了表 14.2 中的内容处理器（handler）方法以外，DefaultHandler 也提供了 DTDHandler, EntityResolver 和 ErrorHandler 接口的方法。你可以使用这些方法来捕获有关 DTD 的信息，可以提供普通实体值，并处理相关错误。

表 14.2 解析器的回调方法

方法	调用
characters	解析字符数据时
endDocument	元素发现结束
endPrefixMapping	发现前缀范围结束
ignorableWhitespace	发现不重要的空格

续表

方法	调用
processingInstruction	发现一个处理指令 (<? ? > 标签)
setDocumentLocator	对通知文档的当前位置的选择回调
skippedEntity	跳过一个实体 (例如外部实体)
startDocument	找到文档的开始
startElement	找到元素的开始
startPrefixMapping	名字空间前缀范围的开头

清单 14.6 一个基于 SAX 的 XML 解析器

```
// XML parser (uses SAX)
// usage java XMLEcho file.xml >outputfile.xml
// Sends report to console

import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;

public class XMLEcho extends DefaultHandler {
    static private Writer out;
    public static void main(String args[]) throws Exception {
        // Use an instance of ourselves as the SAX event handler
        DefaultHandler handler = new XMLEcho();

        // Use the default (non-validating) parser
        SAXParserFactory factory = SAXParserFactory.newInstance();

        // Set up output stream
        out = new OutputStreamWriter(System.out, "UTF8");

        // Parse the input
        SAXParser saxParser = factory.newSAXParser();
        File ifile = new File(args[0]);
        saxParser.parse( ifile, handler );
    }

    private void emit(String s) // write output
        throws SAXException {
        try {
```



```
        out.write(s);
        out.flush();
    } catch (IOException e) {
        throw new SAXException("I/O error", e);
    }
}

private void nl()
throws SAXException    {
    String lineEnd = System.getProperty("line.separator");
    try {
        out.write(lineEnd);

    } catch (IOException e) {
        throw new SAXException("I/O error", e);
    }
}

public void startDocument()
throws SAXException    {
    emit("<?xml version='1.0' encoding='UTF-8'?>");
    System.err.println("Start document");
    nl();
}

public void endDocument()
throws SAXException    {
    try {
        System.err.println("End document");
        nl();
        out.flush();
    } catch (IOException e) {
        throw new SAXException("I/O error", e);
    }
}

public void startElement(String namespaceURI,
                        String sName, // simple name (localName)
                        String qName, // qualified name
                        Attributes attrs)
throws SAXException    {
    String eName = sName; // element name
    if ("".equals(eName)) eName = qName; // namespaceAware = false
    emit("<" + eName);
    System.err.println("TAG: " + eName);
}
```

```

        if (attrs != null) {
            for (int i = 0; i < attrs.getLength(); i++) {
                String aName = attrs.getLocalName(i); // Attr name
                if (".".equals(aName)) aName = attrs.getQName(i);
                emit(" ");
                emit(aName+"=\""+attrs.getValue(i)+"\"");
            }
        }
        emit(">");
    }

    public void endElement(String namespaceURI,
                          String sName, // simple name
                          String qName // qualified name
    )
    throws SAXException {
        String eName = sName; // element name
        if (".".equals(eName)) eName = qName; // namespaceAware = false
        emit("</"+eName+">");
        System.err.println("End tag: " + eName);
    }

    public void characters(char buf[], int offset, int len)
    throws SAXException {
        String s = new String(buf, offset, len);
        if (len!=0&&(len!=1||buf[offset]!='\n'))
            System.err.println("Content: [" + s + "]");
        emit(s);
    }
}

```

当然，除了打印相同的 XML 文件以外，通常你还需要另外做些有意思的工作。你可能要将 XML 转换成另一种格式或者在 XML 数据的基础上创建连接到一个数据库的入口。然而，只要你将各个标签都剥离出来，就可以实现你想要的任何逻辑。

注意：类不能正确地创建那些包含一定入口的 XML 文件。例如，当解析器遇到<gt;实体，它自动地替你的 Java 程序将它转换成正确的字符(">")。但是，当你将那个字符写到新文件中时，它应该转换回原来的实体。如果你没能这样做，其他 XML 解析器将不知道如何形成结果文件。如果你通过 XMLEcho 使用<lt;和<gt;处理一个文件，就可以看到这种情况。输出结果将有一对尖括弧，这是错误的输出。

但是，CDATA 段不使用实体，这样当你处理普通字符或者 CDATA 字符时，就能恰当地处理上述替代情况。不幸的是，这要求有 LexicalHandler 接口，其中 SAX 解析器不需要支持。

解析器的另一个缺点是处理注释。当你处理一个包含注释的输入文件时，注释将不会出现在输出文件里。因为 SAX 解析器在正确解析开始之前的语法分析阶段就将注释抽取出来

了。另外，解决方法是使用可选的 `LexicalHandler` 接口。

`DefaultHandler` 基类没有实现 `LexicalHandler` 接口，这样你不得不自己来实现它。该接口包含下述方法：

- `comment`——解析器发现一个注释。
- `startCDATA`——解析器开始处理未解析的字符数据。
- `endCDATA`——解析器到达了处理的未解析字符的末尾。
- `startEntity`——解析器发现实体的开头。
- `endEntity`——解析器到达实体的末尾。
- `startDTD`——解析器发现文档里的一个 DTD。
- `endDTD`——解析器到达 DTD 的末尾。

`startCDATA` 方法和 `endCDATA` 方法可以用来设置一个标志，以利于其他方法（如 `characters` 方法）可以知道数据是否在 `CDATA` 段中。如果你想处理注释，你可以在 `comment` 段里提供相应代码。记住因为在 `DefaultHandler` 里没有该接口的基本实现，你将不得不为所有这些方法提供函数体，至少是空函数体。

一旦 `LexicalHandler` 接口创建好了，就必须告诉解析器你要使用该接口。`LexicalHandler` 回调并不是解析器的一部分，实际上它是 `XMLReader` 类的一部分，解析器用来读取输入。要设置回调，你需要这样写：

```
saxParser.getXMLReader().setProperty(  
    "http://xml.org/sax/properties/lexical-handler",  
    self);
```

清单 14.7 显示了一个修改过的解析器，它实现了 `LexicalHandler` 接口。当解析器遇到注释时，它向控制台写一个警告消息。只要你想做，你可以轻松地将程序扩展，将注释传递给输出。

清单 14.7 该解析器注意注释是否出现在源文件里

```
// XML parser (uses SAX)  
// usage java XMLEcho1 file.xml >outputfile.xml  
// Sends report to console  
// This version uses a Lexical Handler  
  
import java.io.*;  
import org.xml.sax.*;  
import org.xml.sax.helpers.DefaultHandler;  
import javax.xml.parsers.SAXParserFactory;  
import javax.xml.parsers.ParserConfigurationException;  
import javax.xml.parsers.SAXParser;  
// Import for lex handling  
import org.xml.sax.ext.LexicalHandler;
```

```
public class XMLEcho1 extends DefaultHandler
implements LexicalHandler {
    static private Writer out;
    public static void main(String args[]) throws Exception {
        // Use an instance of ourselves as the SAX event handler
        XMLEcho1 self = new XMLEcho1();
        DefaultHandler handler = self;

        // Use the default (non-validating) parser
        SAXParserFactory factory = SAXParserFactory.newInstance();

        // Set up output stream
        out = new OutputStreamWriter(System.out, "UTF8");

        // Parse the input
        SAXParser saxParser = factory.newSAXParser();
        File ifile = new File(args[0]);
    // Set Lex callback
        saxParser.getXMLReader().setProperty(
            "http://xml.org/sax/properties/lexical-handler",
            self);

        saxParser.parse( ifile, handler );
    }

    private void emit(String s)
        throws SAXException    {
        try {
            out.write(s);
            out.flush();
        } catch (IOException e) {
            throw new SAXException("I/O error", e);
        }
    }

    private void nl()
        throws SAXException    {
        String lineEnd = System.getProperty("line.separator");
        try {
            out.write(lineEnd);

        } catch (IOException e) {
            throw new SAXException("I/O error", e);
        }
    }

    public void startDocument()
```

```

throws SAXException    {
    emit("<?xml version='1.0' encoding='UTF-8'?>");
    System.err.println("Start document");
    nl();
}

public void endDocument()
throws SAXException    {
    try {
        System.err.println("End document");
        nl();
        out.flush();
    } catch (IOException e) {
        throw new SAXException("I/O error", e);
    }
}

public void startElement(String namespaceURI,
                        String sName, // simple name (localName)
                        String qName, // qualified name
                        Attributes attrs)
throws SAXException    {
    String eName = sName; // element name
    if ("".equals(eName)) eName = qName; // namespaceAware = false
    emit("<"+eName);
    System.err.println("TAG: " + eName);
    if (attrs != null) {
        for (int i = 0; i < attrs.getLength(); i++) {
            String aName = attrs.getLocalName(i); // Attr name
            if ("".equals(aName)) aName = attrs.getQName(i);
            emit(" ");
            emit(aName+"=\""+attrs.getValue(i)+"\"");
        }
    }
    emit(">");
}

public void endElement(String namespaceURI,
                      String sName, // simple name
                      String qName // qualified name
                      )
throws SAXException    {
    String eName = sName; // element name
    if ("".equals(eName)) eName = qName; // namespaceAware = false
    emit("</"+eName+">");
    System.err.println("End tag: " + eName);
}

```



```

    }

    public void characters(char buf[], int offset, int len)
    throws SAXException {
        String s = new String(buf, offset, len);
        if (len!=0&&(len!=1||buf[offset]!='\n'))
            System.err.println("Content: [" + s + "]");
        emit(s);
    }

    // The Lex interface

    public void comment(char[] ch, int start, int length)
    throws SAXException {
        System.err.println("Warning: Comment discarded");
    }

    public void startCDATA() throws SAXException {
    }

    public void endCDATA() throws SAXException {
    }

    public void startEntity(String name) throws SAXException {
    }

    public void endEntity(String name) throws SAXException {
    }

    public void startDTD(String name, String publicId, String systemId)
    throws SAXException {
    }

    public void endDTD() throws SAXException {
    }
}

```

清单 14.6 和清单 14.7 中的解析器并不试图使用文档的 DTD（如果输入文档真有 DTD）来验证文档的有效性。如果你想进行验证，需要设置工厂对象，返回一个验证解析器。因此将

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

替换成：

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);
```

你也可以使用 `setNamespaceAware` 来创建工厂，返回能理解名字空间的解析器。当然，你也可以用两个方法调用，得到一个既能理解名字空间，又能进行验证的解析器。

当创建一个校验有效性的解析器时，你解析的文档必须包含一个 DTD，否则你将收到一个警告和一个错误。另外，解析器将使用任意的 DTD 来填充不重要的空白。那意味着输出可能与输入没有相同的空格间隔，除非你修改代码，将不重要的空白传递给程序。

清单 14.8 显示了一个基本的带有验证有效性的 SAX 解析器。要注意一个解析错误会产生对 ErrorHandler 接口的其中一个方法的调用。因为 DefaultHandler 已经在这个接口里实现了，你要做的所有事情就是提供重载方法。通常这些方法所做的处理无非是抛出异常，并将异常接收为参数。

清单 14.8 该解析器通过 DTD 验证 XML 文件的有效性

```
// XML parser (uses SAX)
// usage java XMLValid file.xml >outputfile.xml
// Sends report to console

import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;

public class XMLValid extends DefaultHandler {
    static private Writer out;
    public static void main(String args[]) throws Exception {
        // Use an instance of ourselves as the SAX event handler
        try{
            DefaultHandler handler = new XMLValid();

            SAXParserFactory factory = SAXParserFactory.newInstance();
            factory.setValidating(true);

            // Set up output stream
            out = new OutputStreamWriter(System.out, "UTF8");

            // Parse the input
            SAXParser saxParser = factory.newSAXParser();
            File ifile = new File(args[0]);
            saxParser.parse( ifile, handler );
        } catch (SAXParseException ex) {
            System.out.println("\n** Parsing error"
                + ", line " + ex.getLineNumber()
                + ", uri " + ex.getSystemId());
            System.out.println(" " + ex.getMessage() );

            // Use the contained exception, if any
            Exception x = ex;
```

```

        if (ex.getException() != null)
            x = ex.getException();
        x.printStackTrace();

    } catch (SAXException sxe) {
        // Error generated by this application
        // (or a parser-initialization error)
        Exception x = sxe;
        if (sxe.getException() != null)
            x = sxe.getException();
        x.printStackTrace();

    } catch (ParserConfigurationException pce) {
        // Parser with specified options can't be built
        pce.printStackTrace();

    } catch (IOException ioe) {
        // I/O error
        ioe.printStackTrace();
    }
}

private void emit(String s) throws SAXException {
    try {
        out.write(s);
        out.flush();
    } catch (IOException e) {
        throw new SAXException("I/O error", e);
    }
}

private void nl() throws SAXException {
    String lineEnd = System.getProperty("line.separator");
    try {
        out.write(lineEnd);

    } catch (IOException e) {
        throw new SAXException("I/O error", e);
    }
}

public void startDocument() throws SAXException {
    emit("<?xml version='1.0' encoding='UTF-8'?>");
    nl();
}

public void endDocument() throws SAXException {

```

```

        try {
            nl();
            out.flush();
        } catch (IOException e) {
            throw new SAXException("I/O error", e);
        }
    }

    public void startElement(String namespaceURI,
                            String sName, // simple name (localName)
                            String qName, // qualified name
                            Attributes attrs)
        throws SAXException {
        String eName = sName; // element name
        if (".".equals(eName)) eName = qName; // namespaceAware = false
        emit("<"+eName);
        if (attrs != null) {
            for (int i = 0; i < attrs.getLength(); i++) {
                String aName = attrs.getLocalName(i); // Attr name
                if (".".equals(aName)) aName = attrs.getQName(i);
                emit(" ");
                emit(aName+"=\""+attrs.getValue(i)+"\"");
            }
        }
        emit(">");
    }

    public void endElement(String namespaceURI,
                          String sName, // simple name
                          String qName // qualified name
                          ) throws SAXException {
        String eName = sName; // element name
        if (".".equals(eName)) eName = qName; // namespaceAware = false
        emit("</"+eName+">");
    }

    public void characters(char buf[], int offset, int len)
        throws SAXException {
        String s = new String(buf, offset, len);
        emit(s);
    }

    // ErrorHandler Interface

    // watch for errors

```

```

    public void error(SAXParseException e)
        throws SAXParseException {
        throw e;
    }

    // and warnings
    public void warning(SAXParseException e)
        throws SAXParseException {
        System.out.println("Warning at " + e.getLineNumber() + " " +
            e.getMessage());
    }
}

```

14.1.9 使用 DOM

SAX 解析器策略是边读 XML 文件边进行处理。这样效率比较高。你可以装载大的 XML 文件，不用担心没有足够的存储空间可用。但是，有些程序需要以一种非顺序的方式来处理有关 XML 文件的信息。

因为每个 XML 都包含一个根标签以及它的子标签的多层体系，所以 XML 文档很适合于树形结构。你可以在回调 SAX 解析器的例程里创建一棵树。但是，这个操作很普通，你不用自己来做，JAXP 将为你做这件事。

事实上，读一个 XML 文件并创建一个 DOM 是极其简单的。你必须构造一个 `DocumentBuilder` 对象（其中的处理与包含一个 SAX 解析器非常类似），接着调用一个方法：

```
document = builder.parse( new File("an_xml_file.xml") );
```

现在唯一的问题是没有标准的方法来显示 DOM。Sun 公司对 DOM 解析器的实现中，有一个方法可以导出一个 DOM，但是它并不是对每个解析器都适用。

清单 14.9 中的程序显示了一个 DOM 解析器。它也实现了名为 `walk` 的方法。这个方法以根为 `Node` 的对象（由 `parse` 方法返回的 `Document` 对象，是一种 `Node` 类型）开始。`walk` 方法显示了节点的文本表示，并递归地调用自身来显示子节点。下面是这个程序输出的一个示例：

```

Level 1 Document:#document
  Level 2 Element:book
    Level 3 Comment:#comment
    Level 3 Element:quantity
      Level 4 Text:10
    Level 3 Element:paperback
    Level 3 Element:shelving
      Level 4 Element:shelve
        Level 5 Text:Computer
      Level 4 Element:shelve
        Level 5 Text:Java
      Level 4 Element:shelve

```


Level 5 Text:Programming

walk 方法为 DOM 的结构提供了若干启示。每个 Node 可能有子节点。并且，文本信息总是在 TEXT_NODE 节点里边，这样每个元素节点下可能都有一个 TEXT_NODE 节点。不要期望元素节点自身包含内容数据。例如，在样例文档里，quantity 节点没有包含实际的 quantity (10)。相反，quantity 节点下的 text 节点却有值。

清单 14.9 这个程序使用 DOM 来读 XML

```
// Read XML via a DOM
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import java.io.File;
import java.io.IOException;

import org.w3c.dom.*;

public class DomDemo{
    static Document document;
    static int level=0;

    // convert a node type to a string
    public static String nodeType(Node node) {
        switch (node.getNodeType()) {
            case Node.ATTRIBUTE_NODE:
                return "Attribute";
            case Node.CDATA_SECTION_NODE:
                return "CDATA Section";
            case Node.COMMENT_NODE:
                return "Comment";
            case Node.DOCUMENT_FRAGMENT_NODE:
                return "Doc Fragment";
            case Node.DOCUMENT_NODE:
                return "Document";
            case Node.DOCUMENT_TYPE_NODE:
                return "Document Type";
            case Node.ELEMENT_NODE:
                return "Element";
            case Node.ENTITY_NODE:
                return "Entity";
            case Node.ENTITY_REFERENCE_NODE:
```

```

        return "Entity Reference";
    case Node.NOTATION_NODE:
        return "Notation";
    case Node.PROCESSING_INSTRUCTION_NODE:
        return "Processing Instruction";
    case Node.TEXT_NODE:
        return "Text";
    default:
        return "Unknown";
    }
}

// recursively walk the nodes
public static void walk(Node node) {
    NodeList nodes=node.getChildNodes();
    String val;
    if (node.getNodeType()==Node.TEXT_NODE) {
        val=node.getNodeValue().trim();
        if (val.length()!=0) return; // don't print pure white space
    }
    else
        val=node.getNodeName();
    for (int j=0;j<level;j++) System.out.print("\t");
    System.out.print("Level " + ++level);
    System.out.println(" " + nodeType(node) + ":" + val);
    for (int i=0;i<nodes.getLength();i++)
        walk(nodes.item(i));
    level--;
}

public static void main(String argv[]) throws Exception
{
    DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        // This one line loads the whole document!
        document = builder.parse( new File(argv[0]) );
        walk(document);

    } catch (SAXException sxe) {
        // Error generated during parsing)
        Exception x = sxe;
        if (sxe.getException() != null)
            x = sxe.getException();
        x.printStackTrace();

    } catch (ParserConfigurationException pce) {

```

```

        // Parser with specified options can't be built
        pce.printStackTrace();

        } catch (IOException ioe) {
            // I/O error
            ioe.printStackTrace();
        }
    } // main
}

```

但是,你可能已注意到,清单 14.9 中的程序没有显示属性。尽管有一个 `ATTRIBUTE_NODE` 节点类型,它并没有出现在正常的树形结构里。

如果你想处理属性,就必须在相应元素上调用 `getAttributes` 方法。该方法返回一个 `NamedNodeMap` 对象。这对于 `Node` 对象集合很简单,每个对象与一个属性相对应。这些节点将是 `ATTRIBUTE_NODES` 节点,它们的值也将是属性值。这就是说,在属性下没有文本节点,同样在属性下也没有元素节点。

清单 14.10 显示了修改过的 DOM 程序,输出属性值。其中最大的变化是程序检查每个元素,看看它是否拥有属性。如果有,程序就在处理元素节点之后,再在这些节点上调用 `walk` 方法,但这个调用要在处理任何子节点之前进行。下面是清单 14.10 的输出:

```

Level 1 Document:#document
  Level 2 Element:book
    Level 2 Attribute:Java Network Programming Black Book
    Level 2 Attribute:Al Williams
      Level 3 Comment:#comment
      Level 3 Element:quantity
        Level 4 Text:10
      Level 3 Element:paperback
      Level 3 Element:shelving
        Level 4 Element:shelve
          Level 5 Text:Computer
        Level 4 Element:shelve
          Level 5 Text:Java
        Level 4 Element:shelve
          Level 5 Text:Programming

```

清单 14.10 一个能理解属性的 DOM 程序

```

// Read XML via a DOM
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

```

```
import java.io.File;
import java.io.IOException;

import org.w3c.dom.*;

public class DomDemo{
    static Document document;
    static int level=0;

    // convert a node type to a string
    public static String nodeType(Node node) {
        switch (node.getNodeType()) {
            case Node.ATTRIBUTE_NODE:
                return "Attribute";
            case Node.CDATA_SECTION_NODE:
                return "CDATA Section";
            case Node.COMMENT_NODE:
                return "Comment";
            case Node.DOCUMENT_FRAGMENT_NODE:
                return "Doc Fragment";
            case Node.DOCUMENT_NODE:
                return "Document";
            case Node.DOCUMENT_TYPE_NODE:
                return "Document Type";
            case Node.ELEMENT_NODE:
                return "Element";
            case Node.ENTITY_NODE:
                return "Entity";
            case Node.ENTITY_REFERENCE_NODE:
                return "Entity Reference";
            case Node.NOTATION_NODE:
                return "Notation";
            case Node.PROCESSING_INSTRUCTION_NODE:
                return "Processing Instruction";
            case Node.TEXT_NODE:
                return "Text";
            default:
                return "Unknown";
        }
    }

    // recursively walk the nodes
    public static void walk(Node node) {
        NodeList nodes=node.getChildNodes();
        String val;
        NamedNodeMap attr=null;
        if (node.getNodeType()==Node.TEXT_NODE ||
```

```

        node.getNodeType()==Node.ATTRIBUTE_NODE) {
            val=node.getNodeValue().trim();
            if (val.length()==0) return; // don't print pure white space
        }
        else {
            val=node.getNodeName();
            attr=node.getAttributes();
        }
        for (int j=0;j<level;j++) System.out.print("\t");
        System.out.print("Level " + level+1);
        System.out.println(" " + nodeType(node) + ":" + val);
        if (attr!=null){
            for (int j=0;j<attr.getLength();j++)
                walk(attr.item(j));
        }
        level++;
        for (int i=0;i<nodes.getLength();i++)
            walk(nodes.item(i));
        level--;
    }

    public static void main(String argv[]) throws Exception
    {
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            // This one line loads the whole document!
            document = builder.parse( new File(argv[0]) );
            walk(document);

        } catch (SAXException sxe) {
            // Error generated during parsing)
            Exception x = sxe;
            if (sxe.getException() != null)
                x = sxe.getException();
            x.printStackTrace();

        } catch (ParserConfigurationException pce) {
            // Parser with specified options can't be built
            pce.printStackTrace();

        } catch (IOException ioe) {
            // I/O error
            ioe.printStackTrace();
        }
    } // main
}

```


就像将 XML 文件读进一个 DOM 是可能的一样，创建一个 DOM，接着使用它创建一个输出文件，或者另外处理某些数据是可能的，这些数据并不在一个 XML 文件里，但是处理时认为它们都在一个 XML 文件里。

在快速解决方案一节里的清单 14.14 和清单 14.15 中，你将看到一个这样的例子。只需使用 DOM 的 `createObject` 方法来构造 DOM 树，形成一个根对象。接着使用 `appendChild` 将子节点追加到它们的父节点下。

你也可以使用一个转换器将一个 DOM 转换成一个输出文档。其中的步骤与创建一个解析器时的步骤很相似。首先，你获得一个 `TransformerFactory`。接着向请求工厂获得一个 `Transformer` 对象。你还要通过创建一个 `StreamResult` 对象来设置输出流，从而用它来输出结果。最后，构造一个 `DOMSource` 对象，来封装你要转换成 XML 的 DOM（或者是 DOM 的一部分）。

使用 `Transformer`、`StreamResult` 和 `DOMSource`，你只需简单地调用 `Transformer.transform` 方法，并将它传给 `DOMSource` 和 `StreamResult` 对象。该转换器将合适的 XML 输出到你指定的流里。如果你想看看例子，见快速解决方案一节中的清单 14.15。

14.2 快速解决方案

14.2.1 安装 Java XML 扩展

Sun 知道很多开发人员会下载所有的 XML 的 API 函数和相关库。这就是为何他们创建 JAX 包的原因（<http://java.sun.com/xml/jaxpack.html>）。从这个网页上你可以下载一个包含下述内容的包：

- JAXP——使用含有各种解析器的标准接口来读 XML 文件并解释 XML 文件
- JAXB——允许你将一个 XML 文件写成一个 Java 对象，并将 XML 文件读进一个对象
- JDOM——将 XML 文件转换成 Java 对象树
- JAXM——允许程序互相交换基于 XML 的消息
- JAXR——允许你为其他要寻找 XML 服务的程序发布现存的 XML 服务

要使用 JAXP，你必须将 `jaxp.jar` 以及 `crimson.jar` 文件加到你的 `CLASSPATH` 里边。另外，JAXP 可以提供第三方的解析器接口，你也可以使用 Sun 提供的解析器。如果你使用的是另一种解析器，还需要遵从你使用的解析器的指令。通常，你需要在你的 `CLASSPATH` 里边包含更多的文件，并修改 `javax.xml.parsers.SAXParserFactory` 和 `javax.xml.parsers.DocumentBuilderFactory` 的属性。

14.2.2 在 JSP 里创建 XML

你可以轻易地从一个 JSP 里写一个 XML 输出。关键是使用 `page` 的定向功能来将

contentType 属性设置成 text/xml。当然，你还需要遵守一个 XML 文档的正确的格式。

下面是在 JSP 中创建 XML 的几个步骤：

1. JSP 的第一行要有<?xml ?>标签，这是每个 XML 文件的第一行必备的。
2. 在第二行，使用<%@ page %>定向，设置 contentType 属性为 text/xml。
3. 余下的 XML 文件像通常一样写，你可以使用标准的 JSP 标签（<% %> 或 <%= %>）

输出所有的 XML 标签。

清单 14.11 显示了一个 JSP，它从一个属性文件中读取数据（见清单 14.11），并输出一个 XML 文件。

清单 14.11 该 JSP 将一个属性文件写成一个 XML 文档

```
<?xml version="1.0" ?>
<%@ page contentType="text/xml" %>
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<%
    Properties prop=new Properties();
    FileInputStream fis = new FileInputStream(
        getServletContext().getRealPath(request.getParameter("prop")));
    prop.load(fis);
    String title=prop.getProperty("title");
    String author=prop.getProperty("author");
    int quan=Integer.parseInt(prop.getProperty("quantity"));
    boolean paper=prop.getProperty("paperback").charAt(0)=='t';
%>

<!DOCTYPE book [
    <!ELEMENT book (quantity,(paperback|hardback))>
    <!ELEMENT quantity (#PCDATA)>
    <!ELEMENT paperback EMPTY>
    <!ELEMENT hardback EMPTY>
    <!ATTLIST book title CDATA #REQUIRED >
    <!ATTLIST book author CDATA #REQUIRED >
]>

<book title="<%= title %>" author="<%= author %>">
    <quantity><%= quan %></quantity>
    <% if (paper) { %>
        <paperback/>
    <% } else { %>
        <hardback/>
    <% } %>
</book>
```

清单 14.12 一个示例属性文件，提供给清单 14.11 中的程序使用

```
title=Java Network Programming Black Book
author=Al Williams
quantity=104
paperback=true
hardback=false
```

14.2.3 创建一个解析器

当你想创建一个 JAXP 解析器时，你要遵守下述基本步骤：

1. 通过调用工厂类的一个静态方法 `newInstance`，创建一个工厂类。
2. 使用工厂类的方法来创建一个解析器的新的实例。
3. 调用解析器的 `parse` 方法。

创建一个解析器有两种选择，SAX 解析器允许你串行处理一个 XML 文档。该解析器在探测到 XML 文件的不同元素时，调用你指定的方法。对这种类型的解析器，你可以使用 `SaxParserFactory` 类作为工厂。`newSAXParser` 方法接着创建实际的 `SAXParser` 对象。

在清单 14.6 中可以看到 SAX 解析器的一个样例程序。下面是该程序构造 SAX 解析器的一段代码：

```
// Use the default (non-validating) parser
SAXParserFactory factory = SAXParserFactory.newInstance();
// Parse the input
SAXParser saxParser = factory.newSAXParser();
```

另一种解析 XML 数据的方法是使用 DOM 解析器。其处理是类似的。你要使用 `DocumentBuilderFactory` 作为工厂对象。该工厂的方法会创建一个 `DocumentBuilder` 对象。该对象接着从 XML 输入创建一个 DOM，产生一个 `Document` 对象。在清单 14.9 可以找到一个 DOM 解析器的例子。下面是从该程序创建解析器的一段代码（没有异常处理）：

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

14.2.4 创建一个验证解析器

你可以让一个解析工厂调用工厂对象的 `setValidating` 方法，传给它一个值为 `true` 的参数，产生一个验证解析器。使用验证解析器，输入文档必须有一个相联的 DTD。如果解析器发现与该 DTD 不匹配的语法，它将调用一个指定的错误处理的回调函数。

14.2.5 创建一个理解命名的解析器

如果你想要 XML 解析器理解 XML 名字空间，你可以调用工厂对象的 `setNamespaceAware` 方法，传一个值为 `true` 的参数给它。

14.2.6 使用一个 SAX 解析器

当使用 SAX 解析器时，必须提供一个或者多个回调对象。这些对象实现了具体的接口，解析器在 XML 文件里发现子项时就会调用这些接口。虽然你可以自己来写这些方法，但是扩展 `DefaultHandler` 对象相对比较简单。该对象为 `ContentHandler`, `DTDHandler`, `EntityResolver` 以及 `ErrorHandler` 接口的回调提供了“存根”。

在这些接口当中，`ContentHandler` 接口（见表 14.2）是最有用的。你只需重载你想处理的 `DefaultHandler` 中的方法。例如，这里是探测文档开头（来自清单 14.6 中的一个异常）的程序：

```
public void startDocument()
throws SAXException {
    emit('<?xml version="1.0" encoding="UTF-8"?>');
    System.err.println("Start document");
    nl();
}
```

当然，你必须向解析器报告你的回调对象。通过调用 `parse` 并将你的对象作为第二个参数传递给它。第一个参数是你需要解析的文件：

```
saxParser.parse( ifile, handler );
```

14.2.7 使用 SAX 进行验证

你可能认为创建一个验证解析器就是要求验证一个 XML 文档符合一个 DTD。然而，实际上不只是这样。解析器使用另一个回调接口 `ErrorHandler` 来报告错误。因为 `DefaultHandler` 实现了 `ErrorHandler` 接口，它只是处理发生的所有验证错误。

当然，答案是要重载那些捕获你要处理的错误的方法。接口非常简单，只有三个方法：

- **Error**——当解析器发现一个可恢复的错误时，解析器调用该方法。例如，没能进行 DTD 验证将导致解析器调用该方法。
- **FatalError**——当解析器发现一些严重的事情时，如格式不恰当的 XML 文档；你要接收一个对 `fatalError` 的调用。注意没能通过 DTD 验证的文档仍然可能是格式规范的。
- **Warning**——当解析器发现有些可疑的事时，将其报为一个警告。例如，如果你使用一个验证解析器，并且文档没有包含 DTD，解析器将调用 `warning` 方法。当然，在这里，只要它解析一个元素，它就会调用 `error` 方法。

`ErrorHandler` 接口中的方法接收 `SAXParseException` 对象。你可以将这些异常抛出，这样通常的异常处理就适用了。这些对象包含了关于错误的位置的信息，当报告错误时可能要用到，下面是 `error` 和 `warning` 处理的简单例子：

```
// ErrorHandler Interface

// watch for errors
public void error(SAXParseException e)
```

```

        throws SAXParseException {
            throw e;
        }

        // and warnings
        public void warning(SAXParseException e)
            throws SAXParseException {
            System.out.println("Warning at " + e.getLineNumber() + " " +
                               e.getMessage());
        }

```

注意 `error` 方法实际上抛出了错误（可能是由调用者捕获）。而 `warning` 方法只是打印一条消息并接着解析。

14.2.8 在 XML 文件里创建 DOM

如果你使用 DOM，只需调用 `DocumentBuilder` 对象的 `parse` 方法。它返回一个 `Document` 对象，是 `Node` 类的子类。

```

// This one line loads the whole document!
document = builder.parse( new File(argv[0]) );

```

你可以通过遍历所有节点来检查整个文档。每个 `Node` 对象都拥有方法，允许你用来查找其子节点。例如，你可以调用 `getChildNodes` 方法，它返回一个 `NodeList` 对象。

每个节点都有类型值（使用 `getNodeType` 方法可以找到）。代表标签的元素节点都有文本子节点。这些文本节点包含着标签拥有的数据（使用 `getNodeValue` 方法可以得到）。例如，假设 XML 文件包含如下内容：

```
<pages>40</pages>
```

相应的 DOM 将有一个对应于 `pages` 标签的元素节点。该节点有一个文本子节点，包含着封闭的串值（“40”）。

14.2.9 读属性

在结构化的 DOM 树中没有直接出现的内容是属性。看看这个标签：

```
<width units="in">33</width>
```

如果你使用 `getChildNodes` 方法，并使用给定的 `NodeList` 方法遍历整个树，将看不到 `units` 属性的入口。相反，你必须使用 `getAttribute` 方法来查询每个节点。返回值为 `null` 或者是一个名字节点映射（`NamedNodeMap`）。如果你接收一个有效的 `NamedNodeMap`，它将包含一个或者多个属性节点。你可以调用 `getNodeValue` 方法来查找属性值（`getNodeValue` 方法会告诉你属性名）。

```
NamedNodeMap attr=node.getAttributes();
```

清单 14.13 显示了一个修改过的节点遍历方法，用于读属性值，并采用与显示普通节点相同的方式来显示属性。`walk` 方法检查节点的属性，接着显示当前节点。在显示之后，它在再次调用 `walk` 方法来处理所有子节点之前，会递归调用 `walk` 方法来显示属性值。

清单 14.13 这里的 walk 方法用于处理属性

```
// recursively walk the nodes
public static void walk(Node node) {
    NodeList nodes=node.getChildNodes();
    String val;
    NamedNodeMap attr=null;
    if (node.getNodeType()==Node.TEXT_NODE ||
        node.getNodeType()==Node.ATTRIBUTE_NODE) {
        val=node.getNodeValue().trim();
        if (val.length()==0) return; // don't print pure white space
    }
    else {
        val=node.getNodeName();
        attr=node.getAttributes();
    }
    for (int j=0;j<level;j++) System.out.print("\t");
    System.out.print("Level " + level+1);
    System.out.println(" " + nodeType(node) + ":" + val);
    if (attr!=null){
        for (int j=0;j<attr.getLength();j++)
            walk(attr.item(j));
    }
    level++;
    for (int i=0;i<nodes.getLength();i++)
        walk(nodes.item(i));
    level--;
}
```

14.2.10 构造一个 DOM

有时使用程序创建一个 DOM 比读一个 XML 文件更有用。例如，你可以从一个现存的（非 XML）数据源载入一个 DOM，接着使用 DOM 来创建相应的 XML。

看看清单 14.14 中的程序片段，makeDOM 方法包含一个 DocumentBuilder 对象，但这不是装载一个 XML 文件，而是调用 newDocument 方法来创建一个空的 DOM。

使用这个空的 DOM，调用 createElement 和 appendChild 方法可以构造树。你创建的树将与你从相应的 XML 文件里读出来的树很相像。例如，注意 title 元素，它是一个元素节点，而它的实际文本是一个子元素（文本节点）。

清单 14.14 该方法创建一个 DOM

```
public static void makeDOM() {
    DocumentBuilderFactory factory=
        DocumentBuilderFactory.newInstance ();
    try {
        DocumentBuilder builder=factory.newDocumentBuilder();
```

```
document=builder.newDocument();
Element root =
    (Element) document.createElement("book");
document.appendChild(root);
Element title = (Element) document.createElement("title");
root.appendChild(title);
title.appendChild( document.createTextNode("DeathWorld") );

}

catch (ParserConfigurationException e) {
    e.printStackTrace();
    System.exit(1);
}

}
```

14.2.11 编写一个 DOM

如果你在读或者从其他数据源那里创建 DOM 之前就有了 DOM，你可以通过使用 Transformer 对象将它转换成 XML。构造一个 Transformer 对象比构造一个解析器要简单：

1. 调用静态方法 TransformerFactory.newInstance。
2. 使用 TransformerFactory.newTransformer 方法，返回一个 Transformer 对象的实例。
3. 构造一个包含你要转换的 DOM（可能是一个完整的 DOM 或者是一个较大的 DOM 的子树）的 DOMSource 对象。
4. 构造一个 StreamResult 对象，这是对你要使用的输出流的映射。
5. 调用 Transformer.transform 方法来将 DOM 作为 XML 文件写到流里。

清单 14.15 显示了对程序构造的 DOM 的完整处理过程。makeDOM 方法创建 DOM，程序只是向系统控制台输出 XML。

提示：在编译和运行清单 14.15 中的程序之前，除了 jaxp.jar 和 crimson.jar 文件，你还需要将 xalan.jar 文件添加到你的 CLASSPATH 里。

清单 14.15 你可以使用该程序将 DOM 输出为一个 XML

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import java.io.File;
import java.io.IOException;
```

```
import org.w3c.dom.*;

public class DomXform {
    static Document document;
    private static Element makeElementText(String element,String text) {
        Element e = (Element) document.createElement(element);
        e.appendChild(document.createTextNode(text));
        return e;
    }

    public static void makeDOM() {
        DocumentBuilderFactory factory=
            DocumentBuilderFactory.newInstance ();
        try {
            DocumentBuilder builder=factory.newDocumentBuilder();
            document=builder.newDocument();
            Element root =
                (Element) document.createElement("book");
            document.appendChild(root);
            root.appendChild(makeElementText("title","Deathworld"));
            root.appendChild(makeElementText("author","Harrison"));
            Element shelving =
                (Element) document.createElement("shelving");
            root.appendChild(shelving);
            shelving.appendChild(makeElementText("shelve","scifi"));
            shelving.appendChild(makeElementText("shelve","trilogies"));
            shelving.appendChild(makeElementText("shelve","Harrison"));
        }
        catch (ParserConfigurationException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    static int level=0;

    // convert a node type to a string
    public static String nodeType(Node node) {
        switch (node.getNodeType()) {
            case Node.ATTRIBUTE_NODE:
                return "Attribute";
            case Node.CDATA_SECTION_NODE:
                return "CDATA Section";
            case Node.COMMENT_NODE:
                return "Comment";
        }
    }
}
```

```
        case Node.DOCUMENT_FRAGMENT_NODE:
            return "Doc Fragment";
        case Node.DOCUMENT_NODE:
            return "Document";
        case Node.DOCUMENT_TYPE_NODE:
            return "Document Type";
        case Node.ELEMENT_NODE:
            return "Element";
        case Node.ENTITY_NODE:
            return "Entity";
        case Node.ENTITY_REFERENCE_NODE:
            return "Entity Reference";
        case Node.NOTATION_NODE:
            return "Notation";
        case Node.PROCESSING_INSTRUCTION_NODE:
            return "Processing Instruction";
        case Node.TEXT_NODE:
            return "Text";
        default:
            return "Unknown";
    }
}

public static void main(String argv[]) throws Exception
{
    try {
        // This one line loads the whole document!
        makeDOM();
        // Create a transformer
        TransformerFactory tFactory =
            TransformerFactory.newInstance();
        Transformer transformer = tFactory.newTransformer();

        DOMSource source = new DOMSource(document);
        StreamResult result = new StreamResult(System.out);

        // Write it out
        transformer.transform(source, result);

    } catch (Exception e) {
        e.printStackTrace();
    }
} // main

}
```

清单 14.16 显示了程序的输出，是一个 XML 文件。注意该转换程序没有插入行中断或者缩进标签（人们常用的方法）。另外，XML 是正确的，虽然读起来困难。

清单 14.16 来自清单 14.14 中的程序的 XML 输出

```
<?xml version="1.0" encoding="UTF-8"?>
<book><title>Deathworld</title><author>Harrison</author><shelving>
<shelve>scifi</shelve><shelve>trilogies</shelve><shelve>Harrison</shelve>
</shelving></book>
```


第 15 章 安全性略谈

15.1 深入介绍

你愿意在商场里大声谈论你的支票结余以及你上次体检的结果吗？很可能不。但是，你可以在电话里谈论这些事情，可以在自己的家里谈论这些，你希望有一定的隐私空间，对吗？

当我的小儿儿子出生后，我们屋子里有一个婴儿监视器。它也会连接到邻家的无绳电话的对话。我不想谈其中的细节，但是我发现更多关于我的邻居的情况，比我想要知道的还要多。即使你不使用无绳电话，长距离通话有时可以通过无线电，更不用说是蜂窝电话了。也许你没有使用无线电话，但是你呼叫的人可能正在用无线电话呢。

我的观点是当你知道你没有隐私的时候，隐私就不是问题了。但是，当你对隐私有错觉时，就得加倍小心了。有几个引人注目的案例，在这些案例中，人们普遍认为邮件是隐私，它还是经常骚扰发送者（或者接收者）。从王安到 Oliver North，公众将私人邮件曝光，已经引起好多个公众人物的不幸。

但是安全不只是指邮件方面。就像你在前些章里看到的一样，每次你发送一次密码到 FTP 或者 Telnet 服务器，都要经过 Internet 连接。那样明智吗？是的，婴儿监视器并不偷听 IP 连接，但是它要当心的是有人在你的网络上使用探测器程序或者一个混杂的网卡来窃听你的连接。能够访问你的 ISP（Internet 服务提供商）或者访问你与服务器之间的任何网络中心的人都可以监听你的网络通信（甚至可能在他的空闲时间进行大量的访问）。最后，服务器的系统管理员很可能找到一个方法在你的通信上进行监听。

有多种方法来确保网络通信安全，主要包括以下这些：

- **Wrappers**——像电子邮件或者 FTP 这样的程序，你可以在发送数据之前进行编码，这要假设接收者知道如何去解码。编码并不真的需要任何特殊服务器或者客户端特征，但是有些邮件客户端确实提供了一些简单的方法，让你使用加解密工具来工作。
- **Secure protocols**（安全协议）——有些协议允许你选择可替方法来发送密码。例如，一个服务器可能使用你的密码对一些随机串以秘密的方法进行组合编码。接着，服务器将该串发送到客户端。客户端必须对串进行解码，不是特别安全，再将串返回服务器。这说明客户端不用传输密码就知道密码。在不知道算法和随机串的情况下，你有足够的时间来重新创建密码。
- **Encrypted sockets**（加密套接字）——因为网络使用层来进行操作，在网络协议栈之间插入一个加密层是可能的。例如，理论上，你可以创建加密数据的网卡（对于私

用网络而言)。这将能够阻止你的网络免受普通黑客的入侵,当然那些最顶级的黑客除外。当然,它也会阻止你与其余的 Internet 互操作。更实用的解决方法是在 TCP 层使用一个安全层。安全套接字 (Secure Socket Layer, SSL) 是比较好的选择。很多不安全程序的替代程序也可以获得,例如,不使用 Telnet,很多站点现在使用 SSH (Secure Shell, 安全 Shell),对所有数据加密,不光是对密码进行加密。

- **Data hiding (数据隐藏)——Steganography** 将你的敏感数据以不伤大雅的方式隐藏起来,如一个 GIF 文件或者是 MP3 文件。一般来说该文件显示正常,但是如果你知道数据是如何隐藏的,你就可以恢复它。这可能不太安全,但是它也有优点,那就是它不是很明显的安全消息。如果我向你发送这样的消息 “F\$IDIF DIDFE 98DE#A,” 那么任何人见到这条消息都会猜测它意味着什么东西。如果我向你发送一张带有我的两只狗的图片,有人会猜测到它的实际意思是指我的小虾酱秘方吗? 通常, steganography 使用另一种方法来隐藏数据。

深层次的安全最好留给专家来做,然而,你可以在程序中使用安全特性,无需知道所有的安全细节。

15.1.1 加密技术回顾

我总发现有件有趣的事是高级的专业技术过了 10 年就会变成很平常的技术了。例如,当我使用计算机工作时,程序员要负责将记录穿孔,以适合硬件设备扇区。后来谁还做这样的事呢? 这些事情由操作系统代做了,逻辑记录不必与扇区相符,这种思想也很平常了。网络和图形编程是另两个例子,曾经是某些领域上的专家,现在他们只能算是普通技师。

安全性的背景与之类似。只有核心的专家知道关于安全的复杂细节。但是,安全已经进入普通应用,一般的程序员不用知道所有的安全理论就可以使用它了。

当然,为获得最大安全性,需要有不可破解的代码。假设我在休斯顿,你在三藩。我的公司需要并购三家公司之一,而我想让你知道我们决定并购哪家公司,这样你可以买进那家公司的股票(这种途径当然是非法的,但只是假装那一分钟我们都是穷凶极恶之徒)。在我最后一次进入 Moscone 中心,我们在一个约定的地方,同意为每个公司定一个代码词,分别为 baseball、pizza 和 television。当我向你发送一条消息时,它将只包含上述三个单词之一,这个单词会告诉你要购买哪支股票。

那是不可破解的,不管中途有多少截获的消息。我告诉你我在城里的时候,是多么喜欢你做的 pizza 饼,截获消息的人绝不会想到它的真实含义,除非他知道我们的秘密。

这是非常安全的,但是有一个很大的缺点。它一点也不灵活。如果我的公司决定不并购呢? 或者我们决定并购一家新公司呢? 我们没有代表新公司的代码。在现实生活中,计算机程序想传输任意的文本,如信用卡号、地址以及银行帐户。所以,必须有一个更灵活的方法来对所有串进行编码和解码。

当你为任意串添加编码解码的功能时,你也引入了对加密的攻击的风险。那意味着想知

道我们秘密的人可以使用数学方法或者其他方法来对消息解码。阿兰·图灵在二战期间曾经与他的同伙成功地将德国的 Enigma 传输解密，这也给盟军创造了极大的优势（公平地说，日本使用了称为 Purple 的机器，也被盟军解密了）。这个 Enigma 机器特别复杂，并且它在战后被保密了很多年。但是如果机器可以对某些东西编码，另一台机器可以将它们解码，那就是图灵所做的事，创建第一个真实的计算机来处理解码。

将加密考虑成是一把锁。你愿意使用便宜的自行车锁锁你的家门吗？你银行的贮藏室是如何贮藏你的钱的？没有理由使用 2500 美元的安全系统来锁住 100 美元的自行车。而数据传输案例就不同了。是的，世界上有些国家的计算机拥有能力将你浏览器的 128 位加密数据在几天或者几周内解开。但是他们为何这样做呢？如果你不是妄想在黑塔中计划统治世界的邪恶之徒，那就不用担心了。

编码和解码的一种方法就是使用密码方法。在这种方法里，密钥对你和数据的接收者是可行的。例如，我们可能决定将所有的“A”字符替换成“X”，所有的“B”字符替换成“F”，等等。问题是，如果我知道如何对消息编码，我也知道如何对它进行解码。并且，我们必须有秘密的方法来交换密钥。如果我使用电子邮件向你发送我们的私人代码，那将不太安全。

要克服这种问题，数学上出现了公钥加密。使用公钥加密，有两个密码。第一个是私钥，你将它保存给自己。另一个是公钥，你可以将它传给任何你喜欢的人。任何可以使用公钥的人可以对你用私钥加密过的消息进行加密。数学上是这样，你不能从公钥那里轻易地推算出私钥。

采用公钥相比单个密钥而言，有许多优点。第一，你不必传输私钥给任何人。第二，你可以将你的公钥告诉给任何人。一个使用你的公钥的闯入者也无法读取编过码的消息。

理论上，有些人可以使用你的密钥加密一条消息，接着使用密文来推出密钥，但是它要花去巨大的计算能力。你可以通过增加密钥的位数来增加解密的难度。例如，近来的一个计算机网络小组只使用 22 个小时的时间就破解了 56 位的密码。直到你发现该网络拥有 100,000 台 PC，每秒钟能够试验 2450 亿个密钥，也许那样印象会深些。老实地讲，我的信用卡没有足够的保密功能，值得使用 100,000 台 PC 来获取我的号码。

现在，美国的大多数浏览器使用 128 位密钥，破获这么长的密钥将非常困难，除非有人开发出一种更敏捷的算法来破译密码。

你可以使用公钥的变体来给一个消息授权。假设我使用我的私钥来为一个众所周知的消息加密，并将它发给你。你使用我的公钥将它解密，并确定消息是否正确。你现在更加相信是我发送的消息，因为只有使用我的密钥的人才能解开消息。当然，我也能使用我的私钥加密，再使用你的公钥加密，这样只有你可以读该消息（使用你的私钥），这样你也确信是我发送了该消息（通过应用我的公钥）。

通常公共消息（大家都消息上）实际上是一个消息摘要（一个有效长度检验和）。当你恢复摘要时，你计算你自己的摘要，并比较这两个摘要。它告诉你不仅我发送了该消息，而且消息在传输过程中没有发生改变。

当然，应用长度密钥对于大块数据将非常耗时。为减轻计算机负担，有些系统使用公钥加密来对一个密钥加密。接着密钥对余下的消息进行加密。使用这种方法，你可用较简单的加密算法对大量的消息进行加密。

15.1.2 Java 安全性

如果你发现所有的安全转换容易混淆，不要担心。通常，Java 库会减轻使用安全套接字的“痛苦”。特别地，Java 案例套接字扩展（JSSE）（从 <http://java.sun.com/products/jsse> 可以得到）使得创建一个安全套接字比较简单。如果你正使用 JDK（Java 开发工具包）1.4 版或者更新的版本，它本身就带有 JSSE，否则，可以从 Sun 的网站上下载。

该扩展有三个包：

- `javax.net.ssl`——抽象类定义。
- `javax.net`——用于创建套接字的工厂。
- `com.sun.net.ssl`——Sun 的 JSSE 实现参考。

另外，你可能需要使用 `java.security.cert`，它用于处理安全证书。

不像正规的套接字，它们都有构造函数，你将使用特殊的工厂类来创建安全套接字。并且，你将使用另一种方法来查找工厂类。这种间接构造方法是必要的，因为 JSSE 实际上只是实现类的集合上的一个 shell。尽管你会使用 Sun 的 `com.sun.net.ssl` 包的实现参考，在理论上还是不能从另一种源文件中使用对象。

在清单 15.1 中可以看到一个例子，该程序接受命令行中的主机名作为参数，试图使用 Web 服务器在端口 443 上打开一个安全套接字（https 协议的缺省端口号）。一旦连接上了，程序就发送一个标准的 HTTP1.0 的 GET 请求，并且显示结果。

静态的 `SSLSocketFactory.getDefault` 方法返回套接字工厂。你也可以调用 `createSocket` 方法（不用工厂）创建一个新的套接字，在此之后，程序看起来就像是另一种套接字程序。

虽然程序看起来与普通的套接字程序很相像，你还是会注意到类似的使用正规套接字的程序的执行时间将长得多。那是因为 SSL 套接字在程序可以开始传输数据之前，必须完成一次复杂的握手，来交换密钥和其他信息。

清单 15.1 该程序从一个安全的 Web 服务器那里请求缺省页

```
import java.net.*;
import javax.net.ssl.*;
import java.security.*;
import java.io.*;

public class SSLGet {
    public static void main(String[] args) throws Exception {
        int port=443; // HTTPS uses 443 by default
        String server=args[0];
        SSLSocketFactory factory;
```

```
factory=(SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket socket=(SSLSocket)factory.createSocket(server,port);
Writer out=new OutputStreamWriter(socket.getOutputStream());
out.write("GET https://" + server + "/ HTTP/1.0\r\n\r\n");
out.flush();
BufferedReader in = new BufferedReader(
    new InputStreamReader(socket.getInputStream()));
int c;
while ((c=in.read())!=-1) System.out.print((char)c);
out.close();
in.close();
socket.close();
}
}
```

15.1.3 关于证书

如果你刚开始进行安全工作，那很可能迟早需要一个证书。JDK 中有一个程序，名为 **keytool**，可以帮你创建自己的密钥。该程序允许你创建和维护密钥库。缺省情况下，密钥库是一个文件，但是理论上，它可以是一个数据库、一个可移动的智能卡或者任何其他存储介质。

每个密钥库可以包含两个不同的项：密钥入口，它包含有私钥；还有可信证书入口，它是你（密钥库的主人）信任的公钥。密钥库入口拥有别名，区分大小写，你可以用它来引用入口。**Keytool** 可以使用 X.509 证书（证书的一种标准类型）工作。

你甚至可以创建新的证书，但是没有人会真正信任它们。Web 浏览器以及其他安全程序只会信任一小部分根证书授权（CA）。要想浏览器信任你，你的证书必须经过根 CA 的签名，记住签名可能涉及到一个 CA 链。这样证书可能被我签名，我的证书可能被 Coriolis 签名，而 Coriolis 的证书可能被根 CA 签名，最后的结果与根 CA 直接为你的证书签名是一样的。

表 15.1 显示了证书中的信息。通常，CA 会要求你为证书签名付费。CA 将只让关于你的信息生效。

表 15.1 出现在 X.509 证书中的信息（第 1 版）

域	说明
Version	X.509 版本，keytool 程序将一直考虑到第 1 版，但只创建第 1 版的证书
序列号（Serial Number）	这是一个唯一的序列号
签名算法标识（Signature Algorithm Identifier）	标识了 CA 用于证书签名的算法
签发者名字（Issure Name）	X.500 区分证书签名实体（通常是一个 CA）的名字

续表

域	说明
有效期 (Validity Period)	证书的起止时间和日期
主题名 (Subject Name)	证书识别的公钥实体名
主题公钥信息 (Subject Public Key Information)	实体命名的公钥以及算法标识

最大的 CA 是 VeriSign (www.verisign.com)。在写这本书时, 该 CA 给你一个简单证书 (大多数是 email 签名) 要 14.95 美元。它并购了另一个 CA——Thawte, 它提供的是免费签名证书 (www.thawte.com)。该 CA 会发邮件给你, 核实你的邮件地址是你可以看的那个地址, 并且检验你的邮件地址。但它不会检验你的姓名、电话或者 CA 不去检验的任何其他信息。

表 15.2 (在快速解决方案一节中) 显示了使用 keytool 时的选项。

15.1.4 隐藏数据

我曾经被指责做奇怪的梦, 我想那是真的。一天晚上我梦到通过改变异频雷达收发机的频率来发送秘密数据。为了向外公开, 签名是很普通的, 但如果你知道从哪里可以看到一条消息。当我醒来时, 我想那是一个比较有趣的想法, 于是我做了些研究。

计算起来我做专利应用已经晚了 7000 年的时间, 希腊的 Histiaeus 在一个奴隶的头上写上秘密消息, 等奴隶的头发长出来后, 将奴隶遣走。这就是所谓的信息档案学, 现在它仍然应用得很好。

计算机让档案学发生了革命, 想想所有的文件每天都在 Internet 里穿梭。如果你见到一条消息说 BEGIN PGP, 你知道它使用了 PGP (Pretty Good Privacy) 系统加过密。你可能不知道里边说些什么, 但是你知道发送者和接收者都在交换秘密信息。另一方面, 如果邮件是一幅关于一辆车或者一台计算机的图片, 你可能不会怀疑里边应用了加密。档案程序可以将数据隐藏在图里, 这样图片看起来一样 (或者近乎一样), 但是它包含了一些数据。只有你知道从哪儿可以看到这些数据, 你才知道这些数据。

想象一个黑客侵入你的服务器, 你可以使用假信用卡号来引诱它, 而他忽略了关于你家乡的图片。当然, 隐藏在那些图中的文件可以使用 PGP 或者其他方法进一步加密。但是你必须能在能破解它们之前找到这些文件。

我决定如果必要的话, 可以使用简单的档案学 (可以手工解码) 进行实验。有几个程序可以找到。通过操作单词或者字母组合, 它们可以隐藏文本在其他文本里。但是, 我想有些东西可以集成到 Java I/O 里, 也适合于在 JSP 或者 applet 里使用。事实上, 我想在 Java 的 I/O 系统里集成 steganography (档案学)。

15.1.4.1 Java 的 I/O 系统

使用 Java I/O 工作使我回想起铅工业。拿着两根管子, 放到一起, 然后找到一个适配器

对它们进行校正。可以将大量的管子连起来，如果你有足够多的适配器。

现在的 Java 程序在大多数 I/O 应用中使用了 Reader 和 Writer 类的子类。这些类比老式的流类要好，因为它们处理了 Unicode 映射。如果你有一个流（像 InputStream 和 OutputStream 在控制台中的应用），你可以将它封装到一个 InputStreamReader 或 OutputStreamWriter 对象里，让它的功能与 reader 或者 writer 差不多。

Java 的思想是从一个基本 I/O 开始，再把它封装到其他添加有你需要的功能的对象里。例如，假设你从一个文件开始（使用 FileReader），如果你想从文件中读字符，那就好。但是，你可能想在读的过程中进行缓存，以提高性能。一个缓冲区也让你每次读一整行文本。在这种情况下，你可以将 FileReader 对象封装到 BufferedReader 里。如果你还需要跟踪所在的行号，你可以使用 BufferedReader、LineNumberReader 的子类。另一方面，你可能需要在读完若干字符之后，要将它们压栈，要想有这个功能，你可以使用一个 PushbackReader 类。

混合多个适配器来将管子接在一起是很好接受的。例如，假设你在使用 Socket 类来写一个网络程序，可以使用流（使用 getInputStream 和 getOutputStream 方法）往套接字里读或写。可能使用 InputStreamReader 方法将流转换成一个 reader 对象。接着可能要将 InputStreamReader 传给 LineNumberReader 的构造函数。

如果你是一个 Unix 系统的用户，这可能让你想起在 Unix 里如何执行多任务：你可以使用管道。因此，虽然 `cat foo | more` 显示给你一个文件，但是 `cat foo | sort | more` 显示给你的是相同的文件，只是文件里的行已经排过序。这是很好的一种思路，你很可能想写自己的一个插件模块，它可以改变流的输入或者输出。那就是我的 steganography 对象所做的事，它与一个输出流关联在一起。它可以让你轻松地将数据流隐藏或者暴露出来。

这已经不是新想法了，例如，在 `java.util.zip` 包里已经有类似的对象，执行 zip 文件压缩和解压缩。它也是一种灵活的技术。毕竟，你可以使用 `StringReader` 或者 `StringWriter` 创建一个普通的串作为一个流，因此你可以在串、套接字、管道——任何可以产生流的对象上执行这些过滤操作。还有对象可以将数组转换成流，因此你也可以将过滤器用在数组中。

15.1.4.2 进入 FilterWriter

这些过滤器模块是如此平常，以致于 Java 提供了专门的原型类，能让你使用它的子类来生成它们。特别地，`FilterInputStream`、`FilterOutputStream`、`FilterReader` 和 `FilterWriter` 可以让你分别为 `InputStream`、`OutputStream`、`Reader` 和 `Writer` 对象进行数据预处理。

你不能直接使用这些类。相反，你要创建它们的子类来做你需要的任何专门处理。尽管每个类都有很多功能，你大多数时候还是需要重载 `read` 或 `write` 方法。你的工作比较简单，因为可以写一个函数来处理单个字符，接着按照面向字符的函数定义余下的函数。

例如，考虑清单 15.2 中的类（`UCWriter`）。该对象强制所有输出进入 `writer` 对象，并使用大写。注意只有 `write(int)` 方法拥有真正的代码，另两个 `write` 函数调用该方法来执行转换。当然，在这种情况下，运行时在每个函数中执行不同的转换可能更高效。但是，对于更复杂的转换，可能还是写一次代码，并从其他函数里调用它要好一些。

清单 15.2 这个过滤器强制将流大写

```
// Force a stream to upper case
import java.io.*;

public class UCWriter extends FilterWriter {
    public UCWriter(Writer out) {
        super(out);
    }
    public void write(int c) throws IOException {
        super.write(Character.toUpperCase((char)c));
    }
    public void write(char[] cbuf,int off,int len) throws IOException {
        while (len--!=0) {
            write((int)cbuf[off++]);
        }
    }
    public void write(String str,int off,int len) throws IOException {
        while (len--!=0) {
            write((int)str.charAt(off++));
        }
    }
    public static void main(String args[]) {
        PrintWriter console = new PrintWriter(new
            UCWriter(new
                OutputStreamWriter(System.out)));
        console.println("hello there web techniques!");
        console.flush ();
    }
}
```

当过滤器类想向底层的 writer 写数据时，它使用来自 FilterWriter 的受保护的 out 域。输出中什么也没有出现，一直到该类指定往 out writer 中写数据为止。

该类包含一个 main 例程，该程序将 System.out 转换成 OutputStreamWriter，接着将 writer 与 UCWriter 联系起来。最后，它又将整个 writer 集合与 PrintWriter 的流（不需要格式化）联系起来。最后的结果是一个流，打印系统控制台中的任何内容的大写形式。

15.1.4.3 Steganography

我想用 FilterWriter 类来实现简单的基于文本的 steganography 类，我的想法是我可以将我的类与现存的 Writer 类联系起来，接着对隐藏文本进行编码或者解码。

算法非常简单（见清单 15.3）。编码器读一个纯文本的模板文件。当它输出完文本时，它在单词间带空格，来对你要隐藏的文件中的位进行编码。两个空格表示一个特定状态，一个空格表示一个不同的状态。状态在 1 和 0 之间交替，因此如果两个连续的词后边有两个空格，第一个空格表示 1，第二个空格编码为 0。

清单 15.3 该过滤器使用 steganography 类来隐藏数据

```
// steganography class -- Williams
import java.io.*;

public class Steg extends FilterWriter {
    protected Reader plainmessage; // template
    protected String plainmsgfile; // file name for template
    protected boolean newline;     // need a newline
    protected boolean encoding;    // true if hiding
    protected boolean decphase=false; // decoding phase
    protected int decodech=0;      // decoding character
    protected int decodectr=1;     // current bit decoding
    // when phase is false, 1 space is a 0
    // and 2 spaces is a 1
    // otherwise, invert
    protected boolean phase=false;

    // This is the encoding constructor
    public Steg(String plainmsg,Writer out) throws IOException {
        super(out);
        plainmsgfile=plainmsg;
        plainmessage=new BufferedReader(new FileReader(plainmsgfile));
        encoding=true;
    }

    // This is the decoding constructor
    public Steg(Writer out) {
        super(out);
        encoding=false;
    }

    // Read a character
    // if EOF, return a space and rewind
    protected char readChar() throws IOException {
        int c;
        c=plainmessage.read();
        if (c== -1) {
            plainmessage.close();
            plainmessage=new FileReader(plainmsgfile);
            c=' ';
        }
        return (char)c;
    }

    // Read a word. Skip leading blanks and set newline
    // if you encounter a newline
```

```

protected String readWord() throws IOException {
    StringBuffer wordBuf = new StringBuffer();
    char c;
    newline=false;
    do {
        c=readChar();
        if (c=='\n') newline=true;
    } while (Character.isWhitespace(c)); // skip lead blanks
    do {
        wordBuf.append(c);
        c=readChar();
        if (c=='\n') newline=true;
    } while (!Character.isWhitespace(c)); // read to blank
    return wordBuf.toString();
}

```

```

// Encrypt or decrypt according to flag
public void write(int c) throws IOException {
    // do the "encryption"
    if (encoding) {
        String word,sep;
        boolean bit;
        for (int i=0;i<16;i++) { // unicode is 16 bits
            word=readWord();
            out.write(word);
            if ((c&(1<<i))==0) bit=false; else bit=true;
            if (phase)
                if (bit) sep=" "; else sep=" ";
            else
                if (bit) sep=" "; else sep=" ";
            phase=!phase;
            out.write(sep);
            if (newline) out.write('\n');
        }
    }
    else {
        // decoding
        if (!decphase) {
            if (c==' ') decphase=true;
        }
        else {
            if (c=='\n' || c=='\r') return; // ignore eol
            decphase=false;
            if (c==' '&&!phase || c!=' ' && phase) {
                decodech|=decodectr;
            }
            decodectr<<=1;
        }
    }
}

```



```

        phase=!phase;
        if (decodectr==0x10000) {
            out.write(decodech);
            decodectr=1;
            decodech=0;
        }
    }
}

// This write simply delegates to the first write
public void write(char[] cbuf,int off, int len) throws IOException {
    while (len--!=0) {
        write((int)cbuf[off++]);
    }
}

// This write simply delegates to the first write
public void write(String str,int off, int len) throws IOException {
    while (len--!=0) {
        write((int)str.charAt(off++));
    }
}

// Support for the test main
static void encodetest(String encfile,String inpf) throws Exception {
    FileReader inp=new FileReader(inpf);
    OutputStreamWriter writer = new OutputStreamWriter(System.out);
    Steg stegout=new Steg(encfile,writer);
    do {
        int c=inp.read();
        if (c==-1) break;
        stegout.write(c);
    } while (true);
    stegout.flush();
}

// Support for the test main
static void decodetest(String fn) throws Exception {
    OutputStreamWriter writer = new OutputStreamWriter(System.out);
    Steg stegin=new Steg(writer);
    FileReader fread=new FileReader(fn);
    do {
        int c=fread.read();
        if (c==-1) break;
        stegin.write(c);
    } while (true);
}

```

```
        stegin.flush();
    }

    // Test main -- if two arguments, encode
    // if 1 argument, decode
    // anything else prints help message
    public static void main(String args[]) throws Exception {
        if (args.length==2){
            encodetest(args[0],args[1]);
            System.exit(0);
        }
        if (args.length==1) {
            decodetest(args[0]);
            System.exit(0);
        }
        System.out.println("Usage: Steg template file " +
            "(to encode)\nSteg file (to decode)");
    }
}
```

这个方案不是很完美，因为 Unicode 字符都是 16 位编码，并且压缩比很糟糕（当然，你可以对结果文件进行压缩）。同时，如果在文件开头添加任意的文本，解码就不正确。你可以让解码器搜索特定的关键词，再开始解码（可能另一个关键词用来终止解码）来解决这个问题。编码器接着使用这些特定词构成文件的框架。

该对象包含一个布尔型（encoding），用于跟踪它是不是隐藏文本。有两个不同的构造函数，因此 encoding 状态依赖于哪个构造函数创建了该对象。当编码时，该对象需要一个包含哑文的模板文件，用来隐藏那些待隐藏的文本。

当然，隐藏文件可能需要比模板包含的文本还要多的文本，因此我想在模板文件到达末尾时，重新回卷到开头。不幸的是，FileReader 不支持 reset 方法，因此我不得不临时写一个该方法。

通过添加一个 **BufferedReader** 对象，可以强制 **FileReader** 来支持 mark/reset，但那样我将不得不同意对整个文件进行缓存，这样看起来比较浪费。编码构造函数的参数是模板文本的文件名。接着创建一个其余代码要用的 **FileReader** 对象，当它到达文件末尾时，readChar 方法会关闭这个 **FileReader** 对象，并使用相同的文件名重新创建该对象。尽管这样似乎效率不高，我想它比将整个文件缓存到内存要好一些。

编码器从内部使用 readChar 和 readWord 方法，将模板文件中的单词解析出来。这两个方法也会使调用者确定是否该词的结尾是一个新行符。编码器将这些缓存起来，使得文本看起来更加现实。在解码时，例程忽视任何新行符，防止有回卷行的情况发生（例如，当一个邮件客户端试图对消息格式化的时候）。

15.1.4.4 使用类

在任何 Java 程序（包括 servlet 和 JSP 页）中你可以很容易地使用 Steg 类，我有一个例程 main，这样你可以在命令行里对该类进行测试。准备一个模板文件，名为 template.txt（与你的输入文本相比，它应该很长）。同时创建一个要隐藏的文本文件（我将我的那个文件命名为 input.txt）。接着输入如下命令：

```
java Steg template.txt input.txt >output.txt
```

要恢复隐藏文本，输入下述命令：

```
java Steg output.txt
```

只是想了解一下如何看编码消息，试着对清单 15.4 进行解码，如果你真想知道它里边的内容，试着对该文件运行 Steg 类。如果你发现该消息出现在一个人的收件箱时，估计你不会再想进行这种操作了。

清单 15.4 你能对下述密文解码吗

```
This is not SPAM! You have asked to be kept informed of important
information about how to secure your financial freedom.
Are you tired of working for some one else and never
being better off? Are you tired of having a little extra
month at the end of the money? If
this sounds like you, you should send away for our special
opportunity. If you have a computer and 30 free minutes
a day, you could make a handsome residual income -- one
that you can build to a level you are comfortable with and
then retire. Sounds
too good to be true? That's what I used to think. However,
using this system for only 30 minutes a day has paid
off my house, my car, and allowed me to play golf 5 days a
week. It could happen to you. But it won't if you don't respond
now. How
do you get started? It's easy. You've already taken the first step.
Now all you need to do is hit reply and put I WANT TO WIN in
the subject line. I WANT TO WIN. It's that simple. But
wait! How do you know this is a legitimate offer? What do you have
to lose? If you send me an e-mail, what did it cost? Nothing. If
I send you back a silly picture of a monkey, what does it cost you?
Nothing. But what if I really do have a endless fountain of money
and can show you how to duplicate it? Then not replying will
be what costs you. But
you won't get a silly picture of a monkey. In fact, what you will
get is information on a time tested way to put money in your pocket.
And not just spare change -- real big bucks. Let
chumps work for their pay. You and I know there is a better way. Get
started today. Thanks
for your
```

15.1.4.5 如何保证安全

这种编码方法从攻击的角度来讲，算不上安全。它也不一定要安全。在你可以攻击一个编码方法之前，你必须承认有人在使用。然而，要得到最大限度的安全性，你也可以在隐藏文件之前对文件进行加密。那样，入侵者将可能忽略文件。但是，如果因为某些原因，他们能够解出真实文件，文件同样会被加密。

你可以使用多种方法来改进程序，以抑制攻击。例如，你可以添加起始和终止关键字（像我前边提到过的一样），接着在那些关键字的前后放入哑文。这种情况下，如果不知道关键字，解码难度将更大。你也可以在隐藏数据之前，对数据进行压缩（使用 `java.util.zip` 包中的类）。压缩，通过定义，易从数据中去掉模式（也就是说，增加了平均信息熵），使得统计分析更加困难。如果你知道如何使用 Java 密码库，你可以轻易使用它们，在隐藏数据的同时对数据进行加密。

15.1.4.6 在隐藏之外

虽然该类可能有些用，它也说明了 Java I/O 系统的一个有意思的地方。我们很容易写过过滤器类，插入到现存的流类中，为它们提供更多的特征。例如，你可能会写缓冲类，它们对数据有更多的支持，你的程序也可以用它来提高性能。

因为流可以是网络套接字、串、数组、文件或者管道，你写的过滤器代码将在很多场合都有用。事实上，你可以看到 `java.io` 包中很多预定义的类就是过滤器。

15.2 快速解决方案

15.2.1 创建一个安全的套接字工厂

如果你想创建一个 SSL 套接字，那就首先调用 `SSLSocketFactory` 对象的 `getDefault` 方法。该工厂可以创建安全套接字实例，下面是一些示例程序（引用清单 15.1 中的一个完整例子）：

```
SSLSocketFactory factory;  
factory=(SSLSocketFactory)SSLSocketFactory.getDefault();
```

15.2.2 创建一个安全套接字

有了 `SSLSocketFactory` 类，你可以使用 `createSocket` 方法来创建套接字实例。这个“兜圈子”的方法是必需的，因为 `SSLSocketFactory` 类实际上是对一个提供者类集合的封装，你不用直接与之交互。下面是你将需要的代码（见清单 15.1 中的完整例子）：

```
SSLSocket socket=(SSLSocket)factory.createSocket(server,port);
```

15.2.3 与一个安全的 Web 服务器相连接

也许你想使用安全套接字的是常见原因是为了与一个安全的 Web 服务器相连。缺省情况

下，安全 Web 服务器使用 443 端口。当然，很多站点没有安全服务器，大多数都有一个常规的服务器，端口为 80。

向安全服务器提交与向普通服务器提交非常相似，除了几个地方要特别考虑以外：

- 你必须使用安全套接字。
- 请求必须指定完整的 URL（统一资源定位器）。
- 在大多数情况下，你应该使用 POST，这样送往服务器的变量是不可见的（例如，浏览器中的 URL 行）。

清单 15.5 显示了一个简单的程序，向服务器提交一个域值。程序需要三个命令行参数：

- 主机名（www.al-williams.com），包含着安全服务器（程序假设是 443 端口）。
- 脚本名（例如，ssl/calc.jsp）。
- 要传送的数据（例如，val=100）。程序假设你已经对它进行合适的编码（换句话说，你已经将空格转换成加号，还有其他 URL 编码转换）。

目标文件可能是一个 JSP 或者其他服务器端程序，完成一些简单的任务。我使用一个简单的 JSP 文件，里边只有一行：

```
<%= 2*Integer.parseInt(request.getParameter("val")) %>
```

我将脚本放到名为 calc.jsp 的文件里。接着，我可以使用下面的命令行（当然，你还需对自己的特定服务器及其细节进行调整）对 SSLPost 程序进行测试：

```
java SSLPost www.al-williams.com ssl/calc.jsp val=150
```

结果产生的 Web 页输出将反映正确的答案，码值为 300。

清单 15.5 向安全服务器提交

```
import java.net.*;
import javax.net.ssl.*;
import java.security.*;
import java.io.*;

public class SSLPost {
    public static void main(String[] args) throws Exception {
        int port=443; // HTTPS uses 443 by default
        String server=args[0];
        SSLSocketFactory factory;
        factory=(SSLSocketFactory)SSLSocketFactory.getDefault();
        SSLSocket socket=(SSLSocket)factory.createSocket(server,port);
        Writer out=new OutputStreamWriter(socket.getOutputStream());
        out.write("POST https://" + server + "/" + args[1] +
            " HTTP/1.0\r\n");
        out.write("Content-Type: application/x-www-form-urlencoded\r\n");
        out.write("Content-Length: " + args[2].length() + "\r\n");
        out.write("\r\n");
        out.write(args[2]);
        out.flush();
    }
}
```



```

        BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        int c;
        while ((c=in.read())!=-1) System.out.print((char)c);
        out.close();
        in.close();
        socket.close();
    }
}

```

15.2.4 使用 Steganography

清单 15.3 显示了一个通过扩展 `FilterWriter` 执行 steganography 编码和解码的类。`Steganography` 采用其他数据格式来对数据隐藏进行处理。例如,你可能在一个 GIF 或者 JPEG 文件里隐藏一个文件。清单 15.3 是在文本里隐藏数据。该类在单词间处理空格,使用 0 位或 1 位编码。程序需要一个文本模板文件,该类在单词间使用不同的空格来对文本进行编码产生这个模板文件。你也可以进行逆向处理,使用特殊的间隔将文件转换成纯文本文件。

`FilterWriter` 类对于任何类型的编码器和解码器都很有用。构造函数接受 `Writer` 作为参数,允许你为那个 `Writer` 对象过滤输出。

下面是一个往 `System.out` 里写字符,并通过 `Steg` 对它们进行编码的例子。

```

FileReader inp=new FileReader(inpf); // source of characters
OutputStreamWriter writer = new OutputStreamWriter(System.out);
Steg stegout=new Steg(encfile,writer); // attach Steg to out
do {
    int c=inp.read(); // get characters
    if (c==-1) break;
    stegout.write(c); // filter them through to output
} while (true);
stegout.flush();

```

在这里, `Steg` 构造函数带有两个参数,第一个参数指定带有模板文本的文件,过滤器使用该文件中的词来构造隐藏数据的文本。第二个参数是 `Writer`,你也可以进行反向处理,通过一个参数构造 `Steg` 对象来对文件解码。解码时的构造函数的参数是输出 `Writer`。`Steg` 对象可以探测你调用的是哪个构造函数,并相应修改它的行为。下面是一段示例代码:

```

OutputStreamWriter writer = new OutputStreamWriter(System.out);
Steg stegin=new Steg(writer);
FileReader fread=new FileReader(fn);
do {
    int c=fread.read();
    if (c==-1) break;
    stegin.write(c);
} while (true);
stegin.flush();

```

15.2.5 包含证书

你可以使用 `keytool` 程序来创建自己的测试证书。Sun 文档里有完整的信息。你可以使用 `genkey` 选项来产生一个新的密钥集。存储它们的文件是一个密钥 (keystore) 库, 用来引用密钥的标识符是别名 (alias)。

下面是一个摘录, 里边使用 `keytool` 程序创建一个密钥库, 名为 `awc.keys`, 包含一个名为 `mykey` 的密钥:

```
$ keytool -genkey -alias mykey -keystore awc.keys
Enter keystore password: pontoon
What is your first and last name?
[Unknown]: Al Williams
What is the name of your organizational unit?
[Unknown]: N/A
What is the name of your organization?
[Unknown]: AWC
What is the name of your City or Locality?
[Unknown]: League City
What is the name of your State or Province?
[Unknown]: Texas
What is the two-letter country code for this unit?
[Unknown]: TX
Is CN=Al Williams, OU=N/A, O=AWC, L=League City, ST=Texas, C=TX correct?
[no]: yes

Enter key password for <mykey>
(RETURN if same as keystore password):
```

但是, 像这样的证书不会受到任何人的信任。你需要一个被根 CA 签名的证书, 例如 VeriSign (www.verisign.com) 或者 Thawte (www.thawte.com)。事实上, VeriSign 并购了 Thawte, 但 Thawte 还是自己运营, 还是一家独立的公司。得到免费的测试证书 (以及一定的面向用户证书) 是可能的, 但是对于实际工作中使用的真实证书, 还是希望为其付费。

CA 应该提供关于如何请求一个签名证书的指令。`Keytool` 命令有一个 `-certreq` 命令, 可以用来产生一个签名请求。你通常将会发送该请求到 CA, 它将返回合适的证书, 以供你导入 (见 15.2.7 节 “导入证书”)。

你也可以使用命令行选项来阻止 `keytool` 为你提示任何附加信息。例如, 这个命令行就完全胜任 (即使例子看起来是多行, 它实际上是一个单行命令):

```
keytool -genkey -dname "cn=Joe Jones, ou=NA, o=AWC, c=US" -alias Jones
-keypass arg99a -keystore keys -storepass x99101 -validity 30
```

这里使用密码 `x99101` 产生了密钥库。该密钥库有一个名为 `Jones` 的证书, 带有密码 `arg99a`。证书将有一个普通名 (cn) `Joe Jones`。组织单位 (ou) 是 `NA`, 组织 (o) 是 `AWC`, 国家 (c) 是 `US`。密钥有效期是 30 天。

15.2.6 显示证书

你可以使用 `keytool` 程序的 `-list` 选项来显示一个密钥库或证书。可以单独指定密钥库，列出密钥库中的所有内容。你也可以指定一个别名。例如：

```
keytool -list -keystore awc.keys  
keytool -list -keystore awc.keys -alias mykey
```

你也可以使用 `Keytool` 程序来访问一个存在文件中的证书。

```
keytool -printcert -file alw.cer
```

15.2.7 导入证书

你可以使用 `keytool` 程序的 `-import` 选项将证书导入密钥库，如果你指定了一个现存的别名，`keytool` 假设你使用 CA 证书为一个现存的密钥签名（见“包含证书”一节中的 `-certreq` 的讨论）。如果你指定一个新的别名，`keytool` 将把证书文件看作是你信任的公钥。这里是你将要使用的命令：

```
keytool -import -alias al_williams -file alw.cer
```

15.2.8 导出证书

你可以从密钥库文件中导出一个证书，接着可以将它导入另一个密钥库。下面是一个命令的样例：

```
keytool -export -alias al_williams -file alw.cer
```

附录 A 一些有用的 RFC

如果你需要有关 Internet 上任何内容的详细信息，那么求助于 RFC 是再好不过了。不幸的是，RFC 有成千上万个，你不知道哪个是你需要的。更糟糕的是，很多 RFC 不再有用，它们容易使人混淆。

有些 RFC（像 RFC1000）是其他 RFC 的索引，同时，在线的搜索引擎如 www.faqs.org/rfcs 可能很有帮助。你也可以在 www.ietf.org/iesg/lrfc_index.txt 上找到完整的 RFC 列表，并从 www.rfc-editor.org/rfc.html 检索到文档。

RFC 可以通过邮件收到，邮件地址是 RFC-INFO@ISLEDU。你要得到 RFC 的索引，可以发送下边内容：

HELP: rfc_index

使用下列命令可以得到完整的帮助：

HELP: help

要取到指定的 RFC，可以发送：

Retrieve: RFC

Doc-ID: RFC1000

记住不是所有的 RFC 都是标准。有些甚至比较幽默（例如 RFC1149）。真正的标准文档（以及它们对应的 RFC）出现在表 A.1 中。

表 A.1 Internet 标准文档

标准	标题	作者	日期	作废	RFC 文档
STD0001	Internet Official Protocol Standards	J.Reynolds R.Braden,S.Ginoza	2001.5	RFC2700	RFC2800
STD0002	Assigned Numbers	J.Reynolds,J.Posted	1994.10	RFC1340	RFC1700
STD0003	Requirements for Internet Hosts	R.Braden,Ed.	1989.10		RFC1122, RFC1123
STD0004	Reserved for Router Requirements				
STD0005	Internet Protocol	J.Postel	1981.9		RFC0791, RFC0792, RFC0919, RFC0922, RFC0950, RFC1112

续表

标准	标题	作者	日期	作废	RFC 文档
STD0006	User Datagram Protocol	J.Postel	1980.8		RFC0768
STD0007	Transmission	J.Postel	1981.9		RFC0793
STD0008	Telnet Protocol	J.Postel,J.Reynolds	1983.5		RFC0854, RFC0855
STD0009	File Transfer Protocol	J.Postel,J.Reynolds	1985.10		RFC0959, RFC2228, RFC2640
STD0010	Simple Mail Transfer Protocol	J.Postel	1982.8	RFC788, RFC780, RFC772	RFC0821, RFC2821
STD0011	Standard for the Format of ARPA Internet Text Messages	D.Crocker	1982.8	RFC733	RFC0822, RFC2822
STD0012	Reserved for Network Time Protocol(NTP)				
STD0013	Domain Name System	P.Mockapetris	1987.11		RFC1034, RFC1035
STD0014	Obsolete:Was Mail Routing and Domain System				
STD0015	Simple Network Management Protocol	J.Case,M.Fedor, M.Schoffstall, J.Davin	1990.5		RFC1157
STD0016	Structure of Management Information	M.Rose, K.McCloghrie	1990.5	RFC1065	RFC1155
STD0017	Management Information Base	K.McCloghrie, M.Rose	1991.3	RFC1158	RFC1213
STD0018	Obsolete:Was Exterior Gateway Protocol(RFC904)				
STD0019	NetBIOS Service Protocols	NetBIOS Working Group	1987.3		RFC1001, RFC1002
STD0020	Echo Protocol	J.Postel	1983.5		RFC0862
STD0021	Discard Protocol	J.Postel	1983.5		RFC0863
STD0022	Character Generator Protocol	J.Postel	1983.5		RFC0864

续表

标准	标题	作者	日期	作废	RFC 文档
STD0024	Active Users Protocol	J.Postel	1983.5		RFC0866
STD0025	Daytime Protocol	J.Postel	1983.5		RFC0867
STD0026	Time Server Protocol	J.Postel	1983.5		RFC0868
STD0027	Binary Transmission	J.Postel	1983.5		RFC0856
STD0028	Echo Telnet Option	J.Postel,J.Reynolds	1983.5		RFC0857
STD0029	Suppress Go Ahead Telnet Option	J.Postel,J.Reynolds	1983.5		RFC0858
STD0030	Status Telnet Option	J.Postel,J.Reynolds	1983.5		RFC0859
STD0031	Timing Mark Telnet Option	J.Postel,J.Reynolds	1983.5		RFC0860
STD0032	Extended Options List Telnet Option	J.Postel,J.Reynolds	1983.5		RFC0861
STD0033	Trivial File Transfer Protocol	K.Sollins	1992.7		RFC1350
STD0034	Replaced by STD0056				
STD0035	ISO Transport Service on top of The TCP(Version:3)	M.Rose,D.Cass	1978.5		RFC1008
STD0036	Transmission of IP and ARP over FDDI Networks	D.Katz	1993.1		RFC1390
STD0037	An Ethernet Address Resolution Protocol	D.C. Plummer	1982.11		RFC0826
STD0038	A Reverse Address Resolution Protocol	R.Finlayson,T.Mann, J.Mogul,M.Theimer	1984.6		RFC0903
STD0039	Obsolete:Was BBN Report 1822 (IMP/Host Interface)				
STD0040	Host Access Protocol Specification	Bolt,Beranek and Newman	1993.8		RFC0907
STD0041	Standard for the Transmission of IP Datagrams over Ethernet Networks	C.Hornig	1984.4		RFC0894

续表

标准	标题	作者	日期	作废	RFC 文档
STD0043	Standard for the Transmission of IP Datagrams over IEEE 802 Networks	J.Postel, J.K.Reynolds	1993.8	RFC0948	RFC1042
STD0044	DCN Local-Network Protocols	D.L.Mills	1993.8		RFC0891
STD0045	Internet Protocol on Network System's HYPER-channel:Protocol Specification	K.Hardwick, J.Lekashman	1993.8		RFC1044
STD0046	Transmitting IP Traffic over ARCNET Networks	D.Provan	1993.8	RFC1051	RFC1201
STD0047	Nonstandard for Transmission of IP Datagrams over Serial Lines:SLIP	J.L.Romkey	1993.8		RFC1055
STD0048	Standard for the Transmission of IP Datagrams over NetBIOS Networks	L.J.McLaughlin	1993.8		RFC1088
STD0049	Standard for the Transmission of 802.2 packets over IPX Networks	L.J.McLaughlin	1993.8		RFC1132
STD0050	Definitions of Managed Objects for the Ethernet-like Interface Types	F.Kastenholz	1994.7	RFC1623, RFC1398	RFC1643
STD0051	The Point-to-Point Protocol(PPP)	W.Simpson,Editor	1994.7	RFC1549	RFC1661, RFC1662
STD0052	The Transmisison of IP Datagrams over the SMDS Service	D.Piscitello, J.Lawrence	1991.3		RFC1209
STD0053	Post Office Protocol-Version 3	J.Myers,M.Rose	1996.5	RFC1725	RFC1939
STD0054	OSPF Version 2	J.Moy	1998.4		RFC2328
STD0055	Multiprotocol Interconnect over Frame Relay	C.Brown, A.Malis	1998.9	RFC1490, RFC1294	RFC2427

续表

标准	标题	作者	日期	作废	RFC 文档
STD0057	RIP Version 2 Protocol Applicability Statement	G.Malkin	1994.11		RFC1722
STD0058	Structure of Management Information Version 2(Smiv2)	K.McCloghrie, D.Perkins, J.Schoerwaelder	1999.4	RFC1902	RFC2578, RFC2579
STD0059	Remote Network Monitoring Management Information Base	S.Waldbusser	2000.5	RFC1757	RFC2819
STD0060	SMTP Service Extension for Command Pipelining	N.Freed	2000.9	RFC2197	RFC2920
STD0061	A One-Time Password System	N.Haller,C.Metz, P.Nesser,M.Straw	1998.2	RFC1938	RFC2289

通过参考协议定义的常见名字去找标准会更容易一些。表 A.2 显示了这种关系。

表 A.2 协议中帮助记忆的 STD 文档

助记术语	标题	STD	RFC
ARP	Ethernet Address Resolution Protocol; or, Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware	37	826
CHARGEN	Character Generator Protocol	22	864
Concise-MI	Concise MIB definitions	16	1212
CONF-MIB	Conformance Statements for SMIPv2	58	2580
CONV-MIB	Textual Conventions for SMIPv2	58	2579
DAYTIME	Daytime Protocol	25	867
DISCARD	Discard Protocol	21	863
DOMAIN	Domain names—Implementation and Specification	13	1035
DOMAIN	Domain names—Concepts and Facilities	13	1034
ECHO	Echo Protocol	20	862
ETHER-MIB	Definitions of Managed Objects for the Ethernet-like Interface Types	50	1643
FTP	File Transfer Protocol	9	959
ICMP	Internet Control Message Protocol	5	792

续表

助记术语	标题	STD	RFC
IGMP	Host Extensions for IP Multicasting	5	1112
IP	Internet Protocol	5	791
IP-ARC	Transmitting IP Traffic over ARCNET Networks	46	1201
IP-DC	DCN Local-Network Protocols	44	891
IP-E	Standard for the Transmission of IP Datagrams over Ethernet Networks	41	894
IP-EE	Standard for the Transmission of IP Datagrams over Experimental Ethernet Networks	42	895
IP-FDDI	Transmission of IP and ARP over FDDI Networks	36	1390
IP-FR	Multiprotocol Interconnect over Frame Relay	55	2427
IP-HC	Internet Protocol on Network System's HYPERchannel: Protocol specification	45	1044
IP-IEEE	Standard for the Transmission of IP Datagrams over IEEE 802 Networks	43	1042
IP-IPX	Standard for the Transmission of 802.2 Packets over IPX Networks	49	1132
IP-NETBIOS	Standard for the Transmission of IP Datagrams over NetBIOS Networks	48	1088
IP-SLIP	Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP	47	1055
IP-SMDS	Transmission of IP Datagrams over the SMDS Service	52	1209
IP-WB	Host Access Protocol specification	40	907
MAIL	Standard for the Format of ARPA Internet Text Messages	11	822
MIB-II	Management Information Base for Network Management of TCP/IP-based Internets: MIB-II	17	1213
NETBIOS	Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Detailed Specifications	19	1002
NETBIOS	Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods	19	1001
ONE-PASS	ONE-PASS	61	2289
OSPF2	OSPF Version 2	54	2328
POP3	Post Office Protocol—Version 3	53	1939

续表

助记术语	标题	STD	RFC
PPP-HDLC	PPP in HDLC-like Framing	51	1662
QUOTE	Quote of the Day Protocol	23	865
RARP	Reverse Address Resolution Protocol	38	903
RIP2	RIP Version 2	56	2453
RIP2-APP	RIP Version 2 Protocol Applicability Statement	57	1722
RMON-MIB	Remote Network Monitoring Management Information Base	59	2819
SMI	Structure and Identification of Management Information for TCP/IP-based Internets	16	1155
SMIv2	Structure of Management Information Version 2 (SMIv2)	58	2578
SMTP	Simple Mail Transfer Protocol	10	821
SMTP-Pipe	SMTP Service Extension for Command Pipelining	60	2920
SMTP-SIZE	SMTP Service Extension for Message Size Declaration	10	1870
SNMP	Simple Network Management Protocol (SNMP)	15	1157
TCP	Transmission Control Protocol	7	793
TELNET	Telnet Option Specifications	8	855
TELNET	Telnet Protocol Specification	8	854
TFTP	The TFTP Protocol (Revision 2)	33	1350
TIME	Time Protocol	26	868
TOPT-BIN	Telnet Binary Transmission	27	856
TOPT-ECHO	Telnet Echo Option	28	857
TOPT-EXTOP	Telnet Extended Options: List Option	32	861
TOPT-STAT	Telnet Status Option	30	859
TOPT-SUPP	Telnet Suppress Go Ahead Option	29	858
TOPT-TIM	Telnet Timing Mark Option	31	860
TP-TCP ISO	Transport Services on top of the TCP: Version 3	35	1006
UDP	User Datagram Protocol	6	768
USERS	Active Users	24	866

下面(表 A.3)是最有用或者最有趣(最活跃)的 RFC, 像当初写它们时一样, 同时伴随有它们更新或者作废的信息。我也把这种信息放到中国水利水电出版社网站上, 这样你可

以很容易地搜索到。标准的 RFC 在表中有对应的标题列的标准号。注意有些标准有多个 RFC，这在表中没有反映出来。

表 A.3 有用的 RFC 文档

RFC	标题	作者	日期	更新	作废
0001	Host Software	S. Crocker	1969.4.7		
0002	Host Software	B. Duvall	1969.4.9		
0008	Functional Specifications for the ARPA Network	G. Deloche	1969.5.5		
0009	Host Software	G. Deloche	1969.5.1		
0012	IMP-Host Interface Flow Diagrams	M. Wingfield	1969.8.26		
0013	Zero Text Length EOF Message	V. Cerf	1969.8.20		
0015	Network Subsystem for Time Sharing Hosts	C.S. Carr	1969.9.25		
0017	Some Questions Re: Host-IMP Protocol	J.E. Kresnar	1969.8.27		
0018	IMP-IMP and HOST-HOST Control Links	V. Cerf	1969.9.1		
0020	ASCII Format for Network Interchange	V.G. Cerf	1969.10.16		
0022	Host-host Control Message Formats	V.G. Cerf	1969.10.17		
0023	Transmission of Multiple Control Messages	G. Gregg	1969.10.16		
0024	Documentation Conventions	S.D. Crocker	1969.10.21	RFC0010, RFC0016	RFC0016
0027	Documentation Conventions	S.D. Crocker	1969.12.9	RFC0010, RFC0016, RFC0024	
0028	Time Standards	W.K. English	1970.1.13		
0029	Response to RFC 28	R.E. Kahn	1970.1.19		
0030	Documentation Conventions	S.D. Crocker	1970.2.4	RFC0010, RFC0016, RFC0024, RFC0027	

续表

RFC	标题	作者	日期	更新	作废
0031	Binary Message Forms in Computer	D. Bobrow, W.R. Sutherland	1968.2.1		
0033	New Host-Host Protocol	S.D. Crocker	1970.2.12		RFC0011
0036	Protocol Notes	S.D. Crocker	1970.3.16	RFC0033	
0038	Comments on Network Protocol from NWG/RFC #36	S.M. Wolfe	1970.3.20		
0039	Comments on Protocol Re: NWG/RFC #36	E. Harslem, J.F. Heafner	1970.3.25	RFC0036	
0040	More Comments on the Forthcoming Protocol	E. Harslem, J.F. Heafner	1970.3.27		
0042	Message Data Types	E. Ancona	1970.3.31		
0046	ARPA Network Protocol Notes	E. Meyer	1970.3.17		
0089	Some Historic Moments in Networking	R.M. Metcalfe	1971.1.19		
0093	Initial Connection Protocol	A.M. McKenzie	1971.1.27	RFC0066, RFC0080	
0097	First Cut at a Proposed Telnet Protocol	J.T. Melvin, R.W. Watson	1971.2.15		
0103	Implementation of Interrupt Keys	R.B. Kalin	1971.2.24		
0114	File Transfer Protocol	A.K. Bhushan	1971.4.10		
0128	Bytes	J. Postel	1971.4.21		
0137	Telnet Protocol—a Proposed Document	T.C. O'Sullivan	1971.4.30		
0139	Discussion of Telnet Protocol	T.C. O'Sullivan	1971.5.7	RFC0137	
0141	Comments on RFC 114: A File Transfer Protocol	E. Harslem, J.F. Heafner	1971.4.29	RFC0114	
0147	Definition of a Socket	J.M. Winett	1971.5.7	RFC0129	
0163	Data Transfer Protocols	V.G. Cerf	1971.5.19		
0183	EBCDIC Codes and their Mapping to ASCII	J.M. Winett	1971.7.21		

续表

RFC	标题	作者	日期	更新	作废
0205	NETCRT—a Character Display Protocol	R.T. Braden	1971.8.6		
0206	User Telnet—Description of an Initial Implementation	J.E. White	1971.8.9		
0208	Address Tables	A.M. McKenzie	1971.8.9		
0210	Improvement of Flow Control	W. Conrad	1971.8.16		
0236	Standard Host Names	J. Postel	1971.9.27		RFC0229
0281	Suggested addition to File Transfer Protocol	A.M. McKenzie	1971.12.8	RFC0265	
0318	Telnet Protocols	J. Postel	1972.4.3	RFC0158	
0322	Suggested Telnet Protocol Changes	J. Postel	1972.4.29		
0328	Suggested Telnet Protocol Changes	J. Postel	1972.4.29		
0340	Proposed Telnet Changes	T.C. O'Sullivan	1972.5.15		
0347	Echo Process	J. Postel	1972.5.30		
0348	Discard Process	J. Postel	1972.5.30		
0385	Comments on the File Transfer Protocol	A.K. Bhushan	1972.8.18	RFC0354	
0412	User FTP Documentation	G. Hicks	1972.11.27		
0414	File Transfer Protocol and Further Comments	A.K. Bhushan	1972.12.29	RFC0385	
0429	Character Generator Process	J. Postel	1972.12.12		
0435	Telnet Issues	B. Cosell, D.C. Walden	1973.1.5	RFC0318	
0448	Print Files in FTP	R.T. Braden	1973.2.27		
0461	Telnet Protocol Meeting Announcement	A.M. McKenzie	1973.2.14		
0468	FTP Data Compression	R.T. Braden	1973.3.8		
0480	Host-dependent FTP Parameters	J.E. White	1973.3.8		

续表

RFC	标题	作者	日期	更新	作废
0559	Comments on The New Telnet Protocol and its Implementation	A.K. Bhushan	1973.8.15		
0560	Remote Controlled Transmission and Echoing Telnet Option	D. Crocker, J. Postel	1973.8.13		
0561	Standardizing Network Mail Headers	A.K. Bhushan, K.T. Pogran, R.S. Tomlinson, J.E. White	1973.9.5		
0562	Modifications to the Telnet Specification	A.M. McKenzie	1973.8.28		
0630	FTP Error Code Usage for more Reliable Mail Service	J. Sussmann	1974.4.10		
0640	Revised FTP Reply Codes	J. Postel	1974.6.19	RFC0542	
0652	Telnet Output Carriage-Return Disposition Option	D. Crocker	1974.10.25		
0653	Telnet Output Horizontal Tabstops Option	D. Crocker	1974.10.25		
0654	Telnet Output Horizontal Tab Disposition Option	D. Crocker	1974.10.25		
0655	Telnet Output Formfeed Disposition Option	D. Crocker	1974.10.25		
0656	Telnet Output Vertical Tabstops Option	D. Crocker	1974.10.25		
0657	Telnet Output Vertical Tab Disposition Option	D. Crocker	1974.10.25		
0658	Telnet Output Linefeed Disposition	D. Crocker	1974.10.25		
0659	Announcing Additional Telnet Options	J. Postel	1974.10.18		
0678	Standard File Formats	J. Postel	1974.12.19		
0679	February, 1975, Survey of New-Protocol Telnet servers	D.W. Dodds	1975.2.21		

续表

RFC	标题	作者	日期	更新	作废
0681	Network UNIX	S. Holmgren	1975.3.18		
0697	CWD Command of FTP	J. Lieb	1975.7.14		
0698	Telnet Extended ASCII Option	T. Mock	1975.7.23		
0706	On the Junk Mail Problem	J. Postel	1975.11.8		
0717	Assigned Network Numbers	J. Postel	1976.7.1		
0726	Remote Controlled Transmission and Echoing Telnet Option	J. Postel, D. Crocker	1977.3.8		
0727	Telnet Logout Option	M.R. Crispin	1977.4.27		
0728	Minor Pitfall in the Telnet Protocol	J.D. Day	1977.4.27		
0730	Extensible Field Addressing	J. Postel	1977.5.20		
0732	Telnet Data Entry Terminal Option	J.D. Day	1977.9.12		RFC0731
0734	SUPDUP Protocol	M.R. Crispin	1977.10.7		
0735	Revised Telnet Byte Macro Option	D. Crocker, R.H. Gumpertz	1977.11.3		
0736	Telnet SUPDUP Option	M.R. Crispin	1977.10.31		
0737	FTP Extension: XSEN	K. Harrenstien	1977.10.31		
0738	Time Server	K. Harrenstien	1977.10.31		
0743	FTP Extension: XRSQ/XRCP	K. Harrenstien	1977.12.30		
0748	Telnet Randomly-Lose Option	M.R. Crispin	1978.4.1		
0752	Universal Host Table	M.R. Crispin	1979.1.2		
0753	Internet Message Protocol	J. Postel	1979.3.1		
0756	NIC Name Server—A Datagram-Based Information Utility	J.R. Pickens, E.J. Feinler, J.E. Mathis	1979.1.1		
0759	Internet Message Protocol	J. Postel	1980.8.1		

续表

RFC	标题	作者	日期	更新	作废
0767	Structured Format for Transmission of Multi-media Documents	J. Postel	1980.8.1		
0768	User Datagram Protocol (STD0006)	J. Postel	1980.8.28		
0774	Internet Protocol Handbook: Table of Contents	J. Postel	1980.10.1		RFC0766
0775	Directory Oriented FTP Commands	D. Mankins, D. Franklin, A.D. Owen	1980.12.1		
0779	Telnet Send-Location Option	E. Killian	1981.4.1		
0781	Specification of the Internet Protocol (IP) Timestamp Option	Z. Su	1981.5.1		
0791	Internet Protocol (STD0005)	J. Postel	1981.9.1		RFC0760
0792	Internet Control Message Protocol	J. Postel	1981.9.1		RFC0777
0793	Transmission Control Protocol (STD0007)	J. Postel	1981.9.1		
0794	Pre-emption	V.G. Cerf	1981.9.1	IEN 125	
0795	Service Mappings	J. Postel	1981.9.1		
0796	Address Mappings	J. Postel	1981.9.1		IEN 115
0797	Format for Bitmap Files	A.R. Katz	1981.9.1		
0799	Internet Name Domains	D.L. Mills	1981.9.1		
0813	Window and Acknowledgement Strategy in TCP	D.D. Clark	1982.7.1		
0814	Name, Addresses, Ports, and Routes	D.D. Clark	1982.7.1		
0815	IP Datagram Reassembly Algorithms	D.D. Clark	1982.7.1		
0816	Fault Isolation and Recovery	D.D. Clark	1982.7.1		

续表

RFC	标题	作者	日期	更新	作废
0818	Remote User Telnet Service	J. Postel	1982.11.1		
0826	Ethernet Address Resolution Protocol; or, Converting Network Protocol Addresses to 48.bit Ethernet address for Transmission on Ethernet Hardware (STD0037)	D.C. Plummer	1982.11.1		
0830	Distributed System for Internet Name Service	Z. Su	1982.10.1		
0854	Telnet Protocol Specification (STD0008)	J. Postel, J.K. Reynolds	1983.5.1		RFC0764
0855	Telnet Option Specifications	J. Postel, J.K. Reynolds	1983.5.1		NIC18640
0856	Telnet Binary Transmission (STD0027)	J. Postel, J.K. Reynolds	1983.5.1		NIC15389
0857	Telnet Echo Option (STD0028)	J. Postel, J.K. Reynolds	1983.5.1		NIC15390
0858	Telnet Suppress Go Ahead Option (STD0029)	J. Postel, J.K. Reynolds	1983.5.1		NIC15392
0859	Telnet Status Option (STD0030)	J. Postel, J.K. Reynolds	1983.5.1		RFC0651
0860	Telnet Timing Mark Option (STD0031)	J. Postel, J.K. Reynolds	1983.5.1		NIC16238
0861	Telnet Extended Options: List Option (STD0032)	J. Postel, J.K. Reynolds	1983.5.1		NIC16239
0862	Echo Protocol (STD0020)	J. Postel	1983.5.1		
0863	Discard Protocol (STD0021)	J. Postel	1983.5.1		
0864	Character Generator Protocol (STD0022)	J. Postel	1983.5.1		
0865	Quote of the Day Protocol (STD0023)	J. Postel	1983.5.1		

续表

RFC	标题	作者	日期	更新	作废
0867	Daytime Protocol (STD0025)	J. Postel	1983.5.1		
0868	Time Protocol (STD0026)	J. Postel, K. Harrenstien	1983.5.1		
0869	Host Monitoring Protocol	R. Hinden	1983.12.1		
0872	TCP-on-a-LAN	M.A. Padlipsky	1982.9.1		
0876	Survey of SMTP Implementations	D. Smallberg	1983.9.1		
0879	TCP Maximum Segment Size and Related Topics	J. Postel	1983.11.1		
0885	Telnet End of Record Option	J. Postel	1983.12.1		
0886	Proposed Standard for Message Header Munging	M.T. Rose	1983.12.15		
0887	Resource Location Protocol	M. Accetta	1983.12.1		
0894	Standard for the Transmission of IP Datagrams over Ethernet Networks (STD0041)	C. Hornig	1984.4.1		
0895	Standard for the Transmission of IP Datagrams over Experimental Ethernet Networks (STD0042)	J. Postel	1984.4.1		
0896	Congestion Control in IP/TCP Internetworks	J. Nagle	1984.1.6		
0897	Domain Name System Implementation Schedule	J. Postel	1984.2.1	RFC0881	
0903	Reverse Address Resolution Protocol (STD0038)	R. Finlayson, T. Mann, T. Mann, M. Theimer	1984.6.1		
0906	Bootstrap Loading Using TFTP	R. Finlayson	1984.6.1		

续表

RFC	标题	作者	日期	更新	作废
0908	Reliable Data Protocol	D. Velten, R.M. Hinden, J. Sax	1984.7.1		
0913	Simple File Transfer Simple File Transfer	M. Lottor	1984.9.1		
0917	Internet Subnets	J.C. Mogul	1984.10.1		
0919	Broadcasting Internet Datagrams	J.C. Mogul	1984.10.1		
0920	Domain Requirements	J. Postel, J.K. Reynolds	1984.10.1		
0927	TACACS User Identification Telnet Option	B.A. Anderson	1984.12.1		
0932	Subnetwork Addressing Scheme	D.D. Clark	1985.1.1		
0933	Output Marking Telnet Option	S. Silverman	1985.1.1		
0934	Proposed Standard for Message Encapsulation	M.T. Rose, E.A. Stefferud	1985.1.1		
0935	Reliable Link Layer Protocols	J.G. Robinson	1985.1.1		
0936	Another Internet Subnet Addressing Scheme	M.J. Karels	1985.2.1		
0937	Post Office Protocol: Version 2	M. Butler, J. Postel, D. Chase, J. Goldberger, J.K. Reynolds	1985.2.1		RFC0918
0946	Telnet Terminal Location Number Option	R. Nedved	1985.5.1		
0947	Multi-network Broadcasting within the Internet	K. Lebowitz, D. Mankins	1985.6.1		
0949	FTP Unique-Named Store Command	M.A. Padlipsky	1985.7.1		
0950	Internet Standard Subnetting Procedure	J.C. Mogul, J. Postel	1985.8.1	RFC0792	

续表

RFC	标题	作者	日期	更新	作废
0953	Hostname Server	K. Harrenstien, M.K. Stahl, E.J. Feinler	1985.10.1		RFC0811
0954	NICNAME/WHOIS	K. Harrenstien, M.K. Stahl, E.J. Feinler	1985.10.1		RFC0812
0956	Algorithms for Synchronizing Network Clocks	D.L. Mills	1985.9.1		
0959	File Transfer Protocol (STD0009)	J. Postel, J.K. Reynolds	1985.12.1		RFC0765
0972	Password Generator Protocol	F.J. Wancho	1996.1.1		
0977	Network News Transfer Protocol	B. Kantor, P. Lapsley	1986.2.1		
1000	Request for Comments Reference Guide	J.K. Reynolds, J. Postel	1987.8.1		RFC0999
1011	Official Internet Protocols	J.K. Reynolds, J. Postel	1987.5.1		RFC0991
1042	Standard for the Transmission of IP Datagrams over IEEE 802 Networks (STD0043)	J. Postel, J.K. Reynolds	1988.2.1		RFC0948
1047	Duplicate Messages and SMTP	C. Partridge	1988.2.1		
1049	Content-Type Header Field for Internet Messages	M.A. Sirbu	1988.3.1		
1055	Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP (STD0047)	J.L. Romkey	1988.6.1		
1057	RPC: Remote Procedure Call Protocol Specification: Version 2	Sun Microsystems	1988.6.1		RFC1050
1058	Routing Information Protocol	C.L. Hedrick	1988.6.1		

续表

RFC	标题	作者	日期	更新	作废
1073	Telnet Window Size Option	D. Waitzman	1988.10.1		
1078	TCP Port Service Multiplexer (TCPMUX)	M. Lottor	1988.11.1		
1079	Telnet Terminal Speed Option	C.L. Hedrick	1988.12.1		
1082	Post Office Protocol—Version 3: Extended Service Offerings	M.T. Rose	1988.11.1		
1088	Standard for the Transmission of IP Datagrams over NetBIOS Networks (STD0048)	L.J. McLaughlin	1989.2.1		
1089	SNMP over Ethernet	M.L. Schoffstall, C. Davin, M. Fedor, J.D. Case	1989.2.1		
1090	SMTP on X.25	R. Ullmann	1989.2.1		
1091	Telnet Terminal-type Option	J. VanBokkelen	1989.2.1		RFC0930
1096	Telnet X Display Location Option	G.A. Marcy	1989.3.1		
1097	Telnet Subliminal-Message Option	B. Miller	1989.4.1		
1101	DNS Encoding of Network Names and Other Types	P.V. Mockapetris	1989.4.1	RFC1034, RFC1034,	
1106	TCP Big Window and NAK Options	R. Fox	1989.6.1		
1110	Problem with the TCP Big Window Option	A.M. McKenzie	1989.8.1		
1112	Host Extensions for IP Multicasting	S.E. Deering	1989.8.1	RFC0988, RFC1054	
1118	Hitchhikers Guide to the Internet	E. Krol	1989.9.1		
1122	Requirements for Internet Hosts—Communication Layers (STD0003)	R. Braden, Ed.	1989.10		

续表

RFC	标题	作者	日期	更新	作废
1129	Internet Time Synchronization: The Network Time Protocol	D.L. Mills	1989.10.1		
1141	Incremental Updating of the Internet Checksum	T. Mallory, A. Küllberg	1990.1.1	RFC1071	
1144	Compressing TCP/IP Headers for Low-Speed Serial Links	V. Jacobson	1990.2.1		
1146	TCP Alternate Checksum Options	J. Zweig, C. Partridge	1990.3.1		RFC1145
1149	Standard for the Transmission of IP Datagrams on Avian Carriers	D. Waitzman	1990.4.1		
1153	Digest Message Format	F.J. Wancho	1990.4.1		
1166	Internet Numbers	S. Kirkpatrick, M.K. Stahl, M. Recker	1990.7.1		RFC1117, RFC1062, RFC1020
1176	Interactive Mail Access Protocol—Version 2	M.R. Crispin	1990.8.1		RFC1064
1180	TCP/IP Tutorial	T.J. Socolofsky, C.J. Kale	1991.1.1		
1184	Telnet Linemode Option	D.A. Borman	1990.10.1		RFC1116
1191	Path MTU Discovery	J.C. Mogul, S.E. Deering	1990.11.1		RFC1063
1203	Interactive Mail Access Protocol—Version 3	J. Rice	1991.2.1		RFC1064
1256	ICMP Router Discovery Messages	S. Deering	1991.9.1		
1263	TCP Extensions	S. O'Malley,	1991.10.1		
1288	The Finger User Information Protocol	D. Zimmerman	1991.12		RFC1196, RFC1194, RFC0742

续表

RFC	标题	作者	日期	更新	作废
1305	Network Time Protocol (Version 3) Specification, Implementation	David L. Mills	1992.3		RFC0958, RFC1059, RFC1119
1332	The PPP Internet Protocol Control Protocol (IPCP)	G. McGregor	1992.5		RFC1172
1350	The TFTP Protocol (Revision 2) (STD0033)	K. Sollins	1992.7		RFC0783
1372	Telnet Remote Flow Control Option	C. Hedrick, D. Borman	1992.10		RFC1080
1393	Traceroute Using an IP Option	G. Malkin	1993.1		
1408	Telnet Environment Option	D. Borman, Editor	1993.1		
1411	Telnet Authentication: Kerberos Version 4	D. Borman, Editor	1993.1		
1412	Telnet Authentication: SPX	K. Alagappan	1993.1		
1421	Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures	J. Linn	1993.2		RFC1113
1422	Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Part II: Certificate-Based	S. Kent	1993.2		RFC1114
1423	Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers	D. Balenson	1993.2		RFC1115
1424	Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services	B. Kaliski	1993.2		
1429	Listserv Distribute Protocol	E. Thomas	1993.2		
1436	The Internet Gopher Protocol (a distributed document search and retrieval protocol)	F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, B. Albert	1993.3		

续表

RFC	标题	作者	日期	更新	作废
1459	Internet Relay Chat Protocol	J. Oikarinen, D. Reed	1993.5		
1571	Telnet Environment Option Interoperability Issues	D. Borman	1994.1	RFC1408	
1572	Telnet Environment Option	S. Alexander	1994.1		
1579	Firewall-Friendly FTP	S. Bellovin	1994.2		
1630	Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web	T. Berners-Lee	1994.6		
1652	SMTP Service Extension for 8bit-MIMEtransport	J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker	1994.7		RFC1426
1661	The Point-to-Point Protocol (PPP) (STD0051)	W. Simpson, Editor	1994.7		RFC1548
1663	PPP Reliable Transmission	D. Rand	1994.7		
1681	On Many Addresses per Host	S. Bellovin	1994.8		
1700	Assigned Numbers (STD0002)	J. Reynolds, J. Postel	1994.10		RFC1340
1734	POP3 AUTHentication Command	J. Myers	1994.12		
1736	Functional Recommendations for Internet Resource Locators	J. Kunze	1995.2		
1737	Functional Requirements for Uniform Resource Names	K. Sollins, L. Masinter	1994.12		
1738	Uniform Resource Locators (URL)	T. Berners-Lee, L. Masinter, M. McCahill	1994.12		

续表

RFC	标题	作者	日期	更新	作废
1777	Lightweight Directory Access Protocol	W. Yeong, T. Howes, S. Kille	1995.3		RFC1487
1785	TFTP Option Negotiation Analysis	G. Malkin, A. Harkin	1995.3	RFC1350	
1788	ICMP Domain Name Messages	W. Simpson	1995.4		
1796	Not All RFCs are Standards	C. Huitema, J. Postel, S. Crocker	1995.4		
1808	Relative Uniform Resource Locators	R. Fielding	1995.6	RFC1738	
1834	WHOIS and Network Information Lookup Service, WHOIS++	J. Gargano, K. Weiss	1995.8		
1844	Multimedia E-mail (MIME) User Agent Checklist	E. Huizer	1995.8		RFC1820
1845	SMTP Service Extension for Checkpoint/Restart	D. Crocker, N. Freed, A. Cargille	1995.9		
1846	SMTP 521 Reply Code	A. Durand, F. Dupont	1995.9		
1847	Security Multiparts for MIME: Multipart/Signed and Multipart/Encrypted	J. Galvin, S. Murphy, S. Crocker, N. Freed	1995.10		
1848	MIME Object Security Services	S. Crocker, N. Freed, J. Galvin, S. Murphy	1995.10		
1864	The Content-MD5 Header Field	J. Myers, M. Rose	1995.10		RFC1544
1870	SMTP Service Extension for Message Size Declaration	J. Klensin, N. Freed, K. Moore	1995.11		RFC1653
1873	Message/External-Body Content-ID Access Type	E. Levinson	1995.12		

续表

RFC	标题	作者	日期	更新	作废
1928	SOCKS Protocol Version 5	M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones	1996.3		
1929	Username/Password Authentication for SOCKS V5	M. Leech	1996.3		
1939	Post Office Protocol— Version 3 (STD0053)	J. Myers, M. Rose	1996.5		RFC1725
1945	Hypertext Transfer Protocol—HTTP/1.0	T. Berners-Lee, R. Fielding, H. Frystyk	1996.5		
1957	Some Observations on Implementations of the Post Office Protocol (POP3)	R. Nelson	1996.6	RFC1939	
1990	The PPP Multilink Protocol (MP)	K. Sklower, B. Lloyd, G. McGregor, D. Carr,	1996.8		RFC1717
1991	PGP Message Exchange Formats	D. Atkins, W. Stallings, P. Zimmermann	1996.8		
2015	MIME Security with Pretty Good Privacy (PGP)	M. Elkins	1996.10		
2017	Definition of the URL MIME External-Body Access-Type	N. Freed, K. Moore, A. Cargille	1996.10		
2018	TCP Selective Acknowledgement Options	M. Mathis, J. Mahdavi, S. Floyd, A. Romanow	1996.10		RFC1072
2030	Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI	D. Mills	1996.10		RFC1769

续表

RFC	标题	作者	日期	更新	作废
2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies	N. Freed, N. Borenstein	1996.11		RFC1521, RFC1522, RFC1590
2046	Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types	N. Freed, N. Borenstein	1996.11		RFC1521, RFC1522, RFC1590
2047	MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text	K. Moore	1996.11		RFC1521, RFC1522, RFC1590
2048	Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures	N. Freed, J. Klensin, J. Postel	1996.11		RFC1521, RFC1522, RFC1590
2049	Multipurpose Internet Mail Extensions (MIME) Part Five Conformance Criteria and Examples	N. Freed, N. Borenstein	1996.11		RFC1521, RFC1522, RFC1590
2060	Internet Message Access Protocol—Version 4rev1	M. Crispin	1996.12		RFC1730
2061	IMAP4 Compatibility with IMAP2bis	M. Crispin	1996.12		RFC1730
2062	Internet Message Access Protocol—Obsolete Syntax	M. Crispin	1996.12		
2066	TELNET CHARSET Option	R. Gellens	1997.1		
2075	IP Echo Host Service	C. Partridge	1997.1		
2076	Common Internet Message Headers	J. Palme	1997.2		
2083	PNG (Portable Network Graphics) Specification Version 1.0	T. Boutell	1997.3		
2086	IMAP4 ACL Extension	J. Myers	1997.1		
2087	IMAP4 QUOTA Extension	J. Myers	1997.1		

续表

RFC	标题	作者	日期	更新	作废
2090	TFTP Multicast Option	A. Emberson	1997.2		
2131	Dynamic Host Configuration Protocol	R. Droms	1997.3		
2132	DHCP Options and BOOTP Vendor Extensions	S. Alexander, R. Droms	1997.3		RFC1533
2141	URN Syntax	R. Moats	1997.5		
2145	Use and Interpretation of HTTP Version Numbers	J. C. Mogul, R. Fielding, J. Gettys, H. Frystyk	1997.5		
2227	Simple Hit-Metering and Usage-Limiting for HTTP	J. Mogul, P. Leach	1997.10		
2228	FTP Security Extensions	M. Horowitz, S. Lunt	1997.10	RFC0959	
2311	S/MIME Version 2 Message Specification	S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, L. Repka	1998.3		
2312	S/MIME Version 2 Certificate Handling	S. Dusse, P. Hoffman, B. Ramsdell, J. Weinstein	1998.3		
2315	PKCS #7: Cryptographic Message Syntax Version 1.5	B. Kaliski	1998.3		
2318	The text/css Media Type	H. Lie, B. Bos, C. Lilley	1998.3		
2347	TFTP Option Extension	G. Malkin, A. Harkin	1998.5	RFC1350	RFC1782
2348	TFTP Blocksize Option	G. Malkin, A. Harkin	1998.5	RFC1350	RFC1783
2349	TFTP Timeout Interval and Transfer Size Options	G. Malkin, A. Harkin	1998.5	RFC1350	RFC1784
2368	The Mailto: URL Scheme	P. Hoffman, L. Masinter, J. Zawinski	1998.7	RFC1738, RFC1808	

续表

RFC	标题	作者	日期	更新	作废
2388	Returning Values from Forms: multipart/form-data	L. Masinter	1998.8		
2389	Feature Negotiation Mechanism for the File Transfer Protocol	P. Hethmon, R. Elz	1998.8		
2392	Content-ID and Message-ID Uniform Resource Locators	E. Levinson	1998.8		RFC2111
2414	Increasing TCP's Initial Window	M. Allman, S. Floyd, C. Partridge	1998.9		
2424	Content Duration MIME Header Definition	G. Vaudreuil, G. Parsons	1998.9		
2425	A MIME Content-Type for Directory Information	T. Howes, M. Smith, F. Dawson	1998.9		
2426	vCard MIME Directory Profile	F. Dawson, T. Howes	1998.9		
2428	FTP Extensions for IPv6 and NATs	M. Allman, S. Ostermann, C. Metz	1998.9		
2433	Microsoft PPP CHAP Extensions	G. Zorn, S. Cobb	1998.10		
2437	PKCS #1: RSA Cryptography Specifications Version 2.0	B. Kaliski, J. Staddon	1998.10		RFC2313
2440	OpenPGP Message Format	J. Callas, L. Donnerhake, H. Finney, R. Thayer	1998.11		
2449	POP3 Extension Mechanism	R. Gellens, C. Newman, L. Lundblade	1998.11	RFC1939	
2460	Internet Protocol, Version 6 (IPv6) Specification	S. Deering, R. Hinden	1998.12		RFC1883
2484	PPP LCP Internationalization Configuration Option	G. Zorn	1999.1	RFC2284, RFC1994, RFC1570	

续表

RFC	标题	作者	日期	更新	作废
2516	A Method for Transmitting PPP Over Ethernet (PPPoE)	L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, R. Wheeler	1999.2		
2525	Known TCP Implementation Problems	V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, B. Volz	1999.3		
2554	SMTP Service Extension for Authentication	J. Myers	1999.3		
2557	MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)	J. Palme, A. Hopmann, N. Shelness	1999.3		RFC2110
2577	FTP Security Considerations	M. Allman, S. Ostermann	1999.5		
2581	TCP Congestion Control	M. Allman, V. Paxson, W. Stevens	1999.4		RFC2001
2616	Hypertext Transfer Protocol—HTTP/1.1	R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee	1999.6		RFC2068
2617	HTTP Authentication: Basic Authentication	J. Franks, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, L. Stewart	1999.6		RFC2069
2632	S/MIME Version 3 Certificate Handling	B. Ramsdell, Ed.	1999.6		

续表

RFC	标题	作者	日期	更新	作废
2634	Enhanced Security Services for S/MIME	P. Hoffman, Ed.	1999.6		
2637	Point-to-Point Tunneling Protocol	K. Hamzeh, G. Pall, W. Vertheim, J. Taarud, W. Little, G. Zorn	1999.6		
2646	The Text/Plain Format Parameter	R. Gellens	1999.8	RFC2046	
2659	Security Extensions For HTML	E. Rescorla, A. Schiffman	1999.8		
2660	The Secure HyperText Transfer Protocol	E. Rescorla, A. Schiffman	1990.8		
2774	An HTTP Extension Framework	H. Nielsen, P. Leach, S. Lawrence	2000.2		
2779	Instant Messaging/ Presence Protocol Requirements	M. Day, S. Aggarwal, G. Mohr, J. Vincent	2000.2		
2800	Internet Official Protocol Standards (STD0001)	J. Reynolds, R. Braden, S. Ginoza	2001.5		RFC2700
2810	Internet Relay Chat: Architecture	C. Kalt	2000.4	RFC1459	
2811	Internet Relay Chat: Channel Management	C. Kalt	2000.4	RFC1459	
2812	Internet Relay Chat: Client Protocol	C. Kalt	2000.4	RFC1459	
2813	Internet Relay Chat: Server Protocol	C. Kalt	2000.4	RFC1459	
2821	Simple Mail Transfer Protocol (STD0010)	J. Klensin, Editor	2001.4		RFC0821, RFC0974, RFC1869
2822	Internet Message Format (STD0011)	P. Resnick, Editor	2001.4		RFC0822

续表

RFC	标题	作者	日期	更新	作废
2854	The 'text/html' Media Type	D. Connolly, L. Masinter	2000.6		RFC2070, RFC1980, RFC1942, RFC1867, RFC1866
2898	PKCS #5: Password-Based Cryptography Specification Version 2.0	B. Kaliski	2000.9		
2912	Indicating Media Features for MIME Content	G. Klyne	2000.9		
2913	MIME Content Types in Media Feature Expressions	G. Klyne	2000.9		
2941	Telnet Authentication Option	T. Ts'o, Editor, J. Altman	2000.9		RFC1416
2942	Telnet Authentication: Kerberos Version 5	T. Ts'o	2000.9		
2943	TELNET Authentication Using DSA	R. Housley, T. Horting, P. Yee	2000.9		
2944	Telnet Authentication: SRP	T. Wu	2000.9		
2945	The SRP Authentication and Key Exchange System	T. Wu	2000.9		
2946	Telnet Data Encryption Option	T. Ts'o	2000.9		
2947	Telnet Encryption: DES3 64 bit Cipher Feedback	J. Altman	2000.9		
2948	Telnet Encryption: DES3 64 bit Output Feedback	J. Altman	2000.9		
2949	Telnet Encryption: CAST-128 64 bit Output Feedback	J. Altman	2000.9		
2950	Telnet Encryption: CAST-128 64 bit Cipher Feedback	J. Altman	2000.9		
2951	Telnet Authentication Using KEA and SKIPJACK	R. Housley, T. Horting, P. Yee	2000.9		
2952	Telnet Encryption: DES 64 bit Cipher Feedback	T. Ts'o	2000.9		

续表

RFC	标题	作者	日期	更新	作废
2964	Use of HTTP State Management	K. Moore, N. Freed	2000.10		
2965	HTTP State Management Mechanism	D. Kristol, L. Montulli	2000.10		RFC2109
2980	Common NNTP Extensions	S. Barber	2000.10		
3023	XML Media Types	M. Murata, S. St-Laurent, D. Kohn	2001.1	RFC2048	RFC2376
3030	SMTP Service Extensions for Transmission of Large and Binary MIME Messages	G. Vaudreuil	2000.12		RFC1830
3075	XML-Signature Syntax and Processing	D. Eastlake, J. Reagle, D. Solo	2001.3		
3076	Canonical XML Version 1.0	J. Boyer	2001.3		
3143	Known HTTP Proxy/Caching Problems	I. Cooper, J. Dilley	2001.6		

附录 B 端口的分配

端口号范围从 0 到 65535，从 0 到 1023 的端口号是知名端口。在 Unix 系统里，你可能需要拥有 root 权限才能使用这些端口。除非是故意的，一般不要使用这些端口。1024 到 49151 之间的端口是注册端口。但你可以在自己的程序里使用这些端口，如果你要让自己的程序公开，端口号就可能与注册的端口冲突。最后，剩下的端口，从 49152 以及后边的端口，是免费使用的，只要你喜欢。

表 B.1 显示了你将遇到的最常用的知名端口。要得到一份完整的最新列表，请参考 www.iana.org/assignments/port-numbers 上的 IANA 页面。

表 B.1 常见的端口分配情况

ID 号	端口号/类型	说明
tcpmux	1/tcp	TCP 端口服务多路器
tcpmux	1/udp	TCP 端口服务多路器
rje	5/tcp	远程工作入口
rje	5/udp	远程工作入口
echo	7/tcp	Echo
echo	7/udp	Echo
discard	9/tcp	丢弃
discard	9/udp	丢弃
systat	11/tcp	现存用户
systat	11/udp	现存用户
daytime	13/tcp	时间 (RFC867)
daytime	13/udp	时间 (RFC867)
qotd	17/tcp	日期引用
qotd	17/udp	日期引用
msp	18/tcp	消息发送协议
msp	18/udp	消息发送协议
chargen	19/tcp	字符产生器

续表

ID 号	端口号/类型	说明
chargen	19/udp	字符产生器
ftp-data	20/tcp	文件传输[缺省数据]
ftp-data	20/udp	文件传输[缺省数据]
ftp	21/tcp	文件传输[控制]
ftp	21/udp	文件传输[控制]
ssh	22/tcp	SSH 远程登录协议
ssh	22/udp	SSH 远程登录协议
telnet	23/tcp	Telnet
telnet	23/udp	Telnet
	24/tcp	任意私人邮件系统
	24/udp	任意私人邮件系统
smtp	25/tcp	简单邮件传输
smtp	25/udp	简单邮件传输
msg-icp	29/tcp	MSG ICP
msg-icp	29/udp	MSG ICP
msg-auth	31/tcp	MSG 授权
msg-auth	31/udp	MSG 授权
	35/tcp	任意私人打印服务器
	35/udp	任意私人打印服务器
time	37/tcp	时间
time	37/udp	时间
rap	38/tcp	路由访问协议
rap	38/udp	路由访问协议
rlp	39/tcp	资源定位协议
rlp	39/udp	资源定位协议
nameserver	42/tcp	主机名服务器
nameserver	42/udp	主机名服务器
nicname	43/tcp	别名
nicname	43/udp	别名

续表

ID 号	端口号/类型	说明
ni-ftp	47/udp	NI FTP
tacacs	49/tcp	登录主机协议
tacacs	49/udp	登录主机协议
re-mail-ck	50/tcp	远程邮件检查协议
re-mail-ck	50/udp	远程邮件检查协议
la-maint	51/tcp	IMP 逻辑地址维护
la-maint	51/udp	IMP 逻辑地址维护
domain	53/tcp	域名服务器
domain	53/udp	域名服务器
	57/tcp	任意私人终端访问
	57/udp	任意私人终端访问
	59/tcp	任意私人文件服务
	59/udp	任意私人文件服务
whois++	63/tcp	whois++
whois++	63/udp	whois++
tacacs-ds	65/tcp	TACACS-数据库服务
tacacs-ds	65/udp	TACACS-数据库服务
sql*net	66/tcp	Oracle SQL*NET
sql*net	66/udp	Oracle SQL*NET
bootps	67/tcp	Bootstrap 协议服务器
bootps	67/udp	Bootstrap 协议服务器
bootpc	68/tcp	Bootstrap 协议客户端
bootpc	68/udp	Bootstrap 协议客户端
tftp	69/tcp	简单文件传输
tftp	69/udp	简单文件传输
gopher	70/tcp	Gopher
gopher	70/udp	Gopher
	75/tcp	任意私人拨号服务
	75/udp	任意私人拨号服务

续表

ID 号	端口号/类型	说明
	77/udp	任意私人 RJE 服务
finger	79/tcp	Finger
finger	79/udp	Finger
http	80/tcp	WWW HTTP
http	80/udp	WWW HTTP
hosts2-ns	81/tcp	HOSTS2 名字服务器
hosts2-ns	81/udp	HOSTS2 名字服务器
mfcobol	86/tcp	Micro Focus Cobol
mfcobol	86/udp	Micro Focus Cobol
	87/tcp	任意私人终端链接
	87/udp	任意私人终端链接
kerberos	88/tcp	Kerberos
kerberos	88/udp	Kerberos
npp	92/tcp	网络打印协议
npp	92/udp	网络打印协议
dcp	93/tcp	设备控制协议
dcp	93/udp	设备控制协议
hostname	101/tcp	NIC 主机名字服务器
hostname	101/udp	NIC 主机名字服务器
rtelnet	107/tcp	远程登录服务
rtelnet	107/udp	远程登录服务
pop2	109/tcp	POP 协议-第 2 版
pop2	109/udp	POP 协议-第 2 版
pop3	110/tcp	POP 协议-第 3 版
pop3	110/udp	POP 协议-第 3 版
sunrpc	111/tcp	SUN 远程过程调用
sunrpc	111/udp	SUN 远程过程调用
auth	113/tcp	授权服务
auth	113/udp	授权服务

续表

ID 号	端口号/类型	说明
sqlserv	118/udp	SQL 服务
nntp	119/tcp	网络新闻传输协议
nntp	119/udp	网络新闻传输协议
ntp	123/tcp	网络时间协议
ntp	123/udp	网络时间协议
pwdgen	129/tcp	密码产生器协议
pwdgen	129/udp	密码产生器协议
statsrv	133/tcp	统计服务
statsrv	133/udp	统计服务
epmap	135/tcp	DCE 终端解析
epmap	135/udp	DCE 终端解析
Netbios-ns	137/tcp	Netbios 名字服务
Netbios-ns	137/udp	Netbios 名字服务
Netbios-dgm	138/tcp	Netbios 数据报服务
Netbios-dgm	138/udp	Netbios 数据报服务
Netbios-ssn	139/tcp	Netbios 会话服务
Netbios-ssn	139/udp	netbios 会话服务
imap	143/tcp	Internet 消息访问协议
imap	143/udp	Internet 消息访问协议
Sql-net	150/tcp SQL	SQL-NET
Sql-net	150/udp SQL	SQL-NET
sqlsrv	156/tcp	SQL 服务
sqlsrv	156/udp	SQL 服务
snmp	161/tcp	SNMP
snmp	161/udp	SNMP
snmptrap	162/tcp	SNMPTRAP
snmptrap	162/udp	SNMPTRAP
print-srv	170/tcp	Network PostScript
print-srv	170/udp	Network PostScript

续表

ID 号	端口号/类型	说明
irc	194/udp	Internet 中继聊天协议
ipx	213/tcp	IPX
ipx	213/udp	IPX
imap3	220/tcp	交互邮件访问协议 v3
imap3	220/udp	交互邮件访问协议 v3
fln-spx	221/tcp	带有 SPX 授权的 Berkeley rlogind
fln-spx	221/udp	带有 SPX 授权的 Berkeley rlogind
rsh-spx	222/tcp	带有 SPX 授权的 Berkeley rshd
rsh-spx	222/udp	带有 SPX 授权的 Berkeley rshd
set	257/tcp	安全电子事务
set	257/udp	安全电子事务
http-mgmt	280/tcp	http-mgmt
http-mgmt	280/udp	http-mgmt
ups	401/tcp	不可中断电源供应
ups	401/udp	不可中断电源供应
https	443/tcp	基于 TLS/SSL 的 http 协议
https	443/udp	基于 TLS/SSL 的 http 协议
biff	512/udp	邮件系统中用来通知用户
login	513/tcp	远程登录
who	513/udp	维护显示 who 的数据库
shell	514/tcp	cmd
syslog	514/udp	
printer	515/tcp	spooler
printer	515/udp	spooler
talk	517/tcp	像 tenex link
talk	517/udp	像 tenex link
ntalk	518/tcp	
ntalk	518/udp	

续表

ID 号	端口号/类型	说明
utime	519/udp	unixtime
efs	520/tcp	扩展文件名服务器
router	520/udp	本地路由处理
ripng	521/tcp	ripng
ripng	521/udp	ripng
ulp	522/tcp	ULP
ulp	522/udp	ULP
nep	524/tcp	NCP
nep	524/udp	NCP
timed	525/tcp	时间服务器
timed	525/udp	时间服务器
tempo	526/tcp	新日期
Irc-serv	529/tcp	IRC -SERV
irc-serv	529/udp	IRC-SERV
conference	531/tcp	聊天
conference	531/udp	聊天
netnews	532/tcp	读新闻
netnews	532/udp	读新闻
netwall	533/tcp	紧急广播
netwall	533/udp	紧急广播
uucp	540/tcp	uucpd
uucp	540/udp	uucpd
uucp-rlogin	541/tcp	uucp-rlogin
uucp-rlogin	541/udp	uucp-rlogin
new-rwho	550/tcp	new-who
new-rwho	550/udp	new-who
cybercash	551/tcp	cybercash
cybercash	551/udp	cybercash
nntp	563/tcp	基于 TLS/SSL 的 nntp 协议

续表

ID 号	端口号/类型	说明
whoami	565/tcp	whoami
whoami	565/udp	whoami
sntp-heartbeat	580/tcp	SNTP HEARTBEAT
sntp-heartbeat	580/udp	SNTP HEARTBEAT
doom	666/tcp	Doom Id 软件
doom	666/udp	Doom Id 软件
corba-iiop	683/tcp	CORBA IIOP
corba-iiop	683/udp	CORBA IIOP
corba-iiop-ssl	684/tcp	CORBA IIOP SSL
corba-iiop-ssl	684/udp	CORBA IIOP SSL
uuidgen	697/tcp	UUIDGEN
uuidgen	697/udp	UUIDGEN
kerberos-adm	749/tcp	kerberos 管理
kerberos-adm	749/udp	kerberos 管理
kerberos-iv	750/udp	kerberos 第 iv 版

Java学习群

72030155

进群可以免费获取视频教程以及每日免费听老师讲课